SFA Modernization Partner

United States Department of Education

Student Financial Assistance

Integrated Technical Architecture

Detailed Design Document

Volume 3 – Enterprise Application Integration Architecture

Task Order #16

Deliverable # 16.1.2

October 13, 2000

Table of Contents

List of Figures

List of Tables

1 Introduction

The Enterprise Application Integration (EAI) system is part of the Executive Architecture for the Department of Education (DOE) Student Financial Assistance (SFA) system as part of the Modernization Blueprint. EAI is a set of technology services that enables the sharing of processes and data of disparate systems to support end-to-end business processes. The EAI Architecture enables the many "stovepipe" applications to exchange information via common, reusable methods and infrastructure.

EAI will allow the SFA program for the Department of Education to integrate new web-based applications with existing back-end systems, while at the same time, providing a means to migrate away from reliance upon existing legacy systems.

1.1. Objectives

The objectives of the EAI section of the document is to assist in developing the Development, Execution, and Operations architectures for the initial release of the Integrated Technical Architecture (ITA).

1.2. Scope

The scope of the EAI section is to provide an overview of the EAI application design, and provide application-programming guidelines for EAI development and design.

2 EAI Architecture Overview

EAI provides capabilities that will allow for the integration of web-based applications, the Data Warehouse environment, commercial-off-the-shelf (COTS) packages, and existing legacy systems within the SFA technical environment.

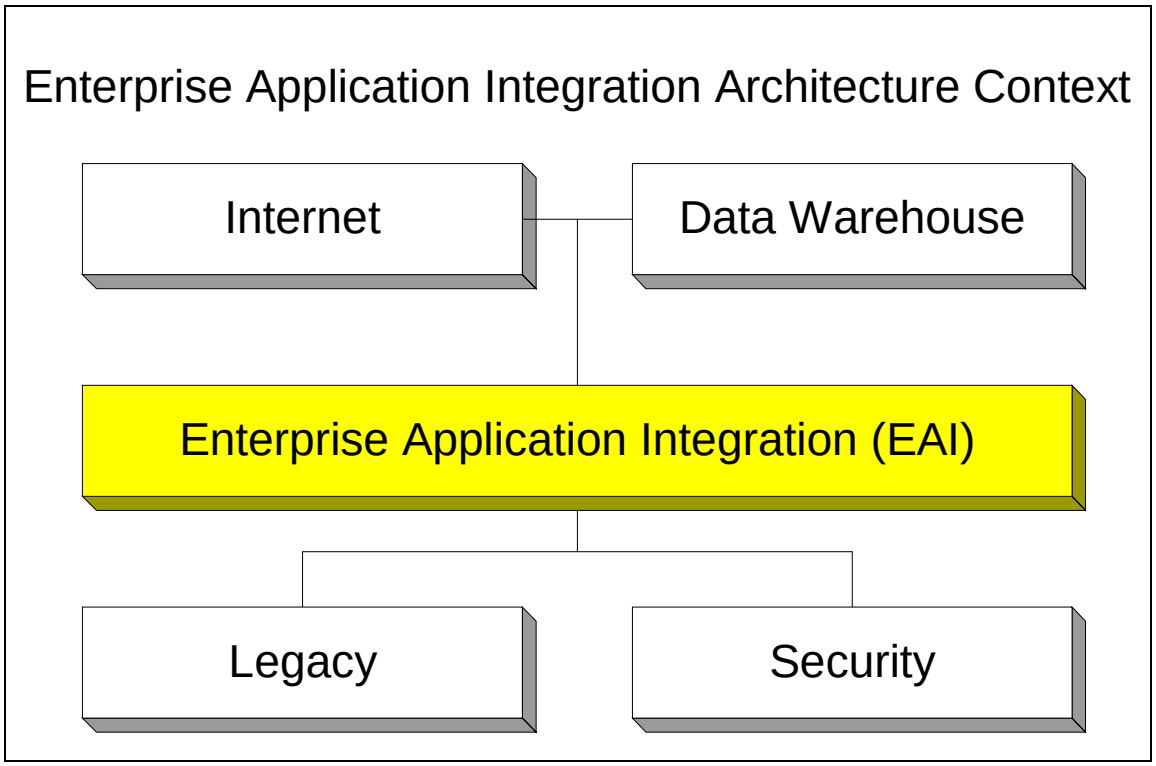

Figure 1 – EAI Architecture Context

The EAI architecture provides the following technical services:

- Communications Middleware
- Transformation and Formatting
- Application Connectivity
- Business Process Management

For a detailed view of the EAI technical services, refer to Deliverable 4.1.2 – Recommended Application Architecture Standards.

2.1. Communication Middleware

The Communications Middleware component provides the architecture that implement various messaging models and route messages according to message content and context.

These services provide the connection among disparate resources, as well as security, queuing, and the functionality to reconcile network protocol differences.

Communications middleware:

- Directs the flow of messages among applications
- Supports both synchronous and asynchronous communications
- Routes messages to applications based on message subject and / or content
- Provides services via message brokers, Object Request Brokers (ORB), or message queues

2.2. Transformation and Formatting

The Transformation and Formatting layer is responsible for the conversion of data and message content and syntax to reconcile the differences between data from multiple heterogeneous systems and data sources. This layer is responsible for maintaining the information structure of the messages passed between systems and their meaning in a format that can be comprehended by another application.

The transformation and formatting layer supports:

- Message protocol and format transformation
- Syntactic translation of one data set into another. (Example: translation of date formats, 01 Aug 1999 -> 19990801)
- Semantic translation of data based on underlying data definitions or meaning. (Example: conversion from the English system to the metric system)

2.3. Application Connectivity

The Application Connectivity layer provides reusable, non-invasive connectivity with legacy systems and external databases.

The Application Connectivity layer provides:

- Pre-built application adapters to access legacy systems and databases
- Connection managed to and from source application
- Connectors to common technologies such as ORBs, support for Common Object Request Broker Architecture (CORBA), Enterprise Java Beans (EJB), etc.

2.4. Business Process Management

The Business Process Management layer is responsible for the definition and management of cross-application business processes across the enterprise and between enterprises. These services enable the communication not just of data, but also of the business process context of the data being sent to another application.

Business Process Management provides:

- Centralized visibility and control of multi-step business processes traversing multiple applications
- Real-time analysis capabilities
- Workflow-like coordination of multi-step processes
- Transactional control
- Process state information maintained to support rollback processes
- Graphical tools and metadata to define processes and rules

2.5. EAI Architecture Solution

The EAI Architecture solution for the Integrated Technical Architecture (TA)is based on the International Business Machines (IBM) MQSeries product family. Each of the high-level EAI technical services will be provided by a specific MQSeries application or component.

- Communications Middleware → MQSeries Messaging
- Transformation and Formatting → MQSeries Integrator.
- Application Connectivity → Adapters and MQSeries Bridges
- Business Process Management → MQSeries Workflow

The diagram below depicts the EAI Architecture and the relationship between the MQSeries products. The MQSeries Adapters and Bridges are used as interfaces with external systems and are not shown as part of this diagram.

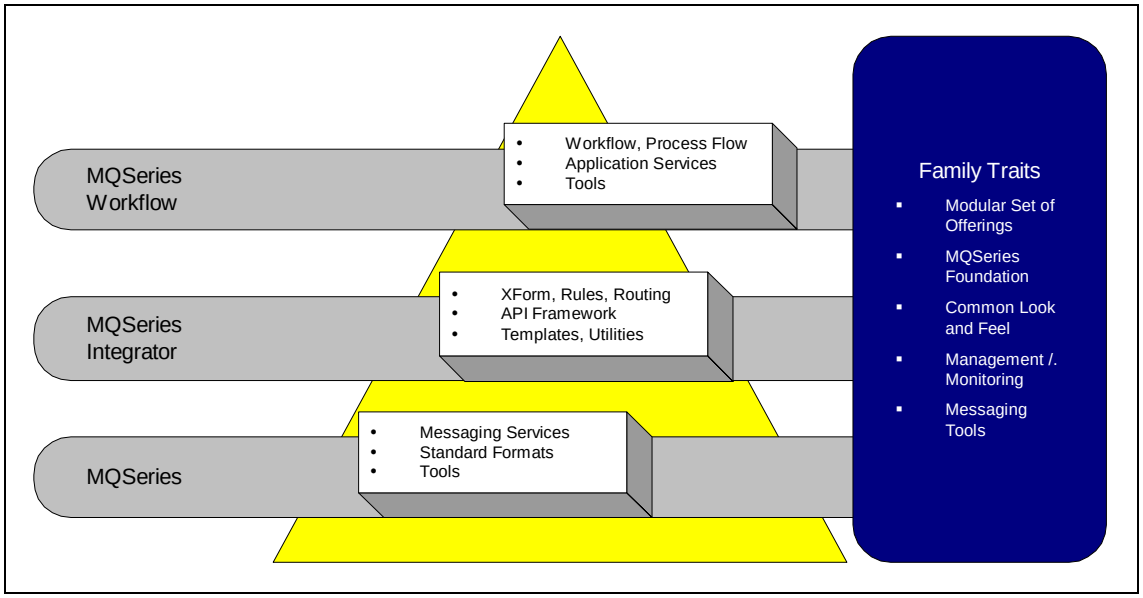

Figure 2 – EAI MQSeries Product

3 EAI Technical Design

This section of the document contains the EAI design specifications for the Technical Architecture. The EAI components are part of the overall ITA Execution Architecture.

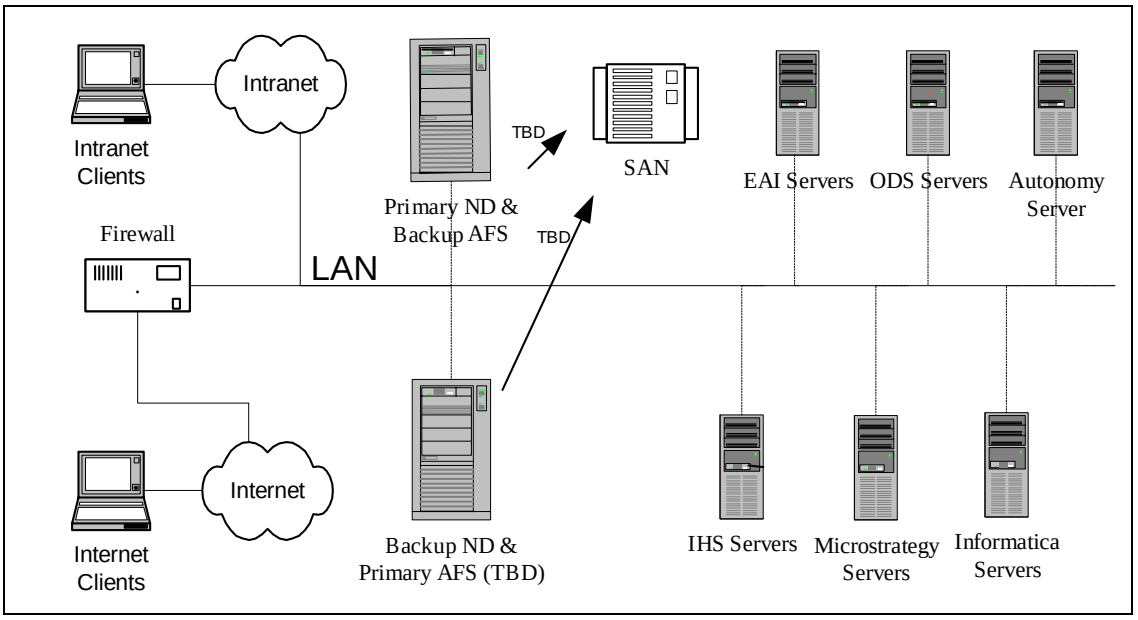

Figure 3 – EAI Technical Design Architecture

In the initial release of the ITA, the EAI application will source and supply data to/from external systems such as Internet applications and existing legacy systems. The EAI application will have the capability to translate data through programmable, user defined rules. The EAI application will help facilitate the routing of data to various datastores, and facilitates querying such data.

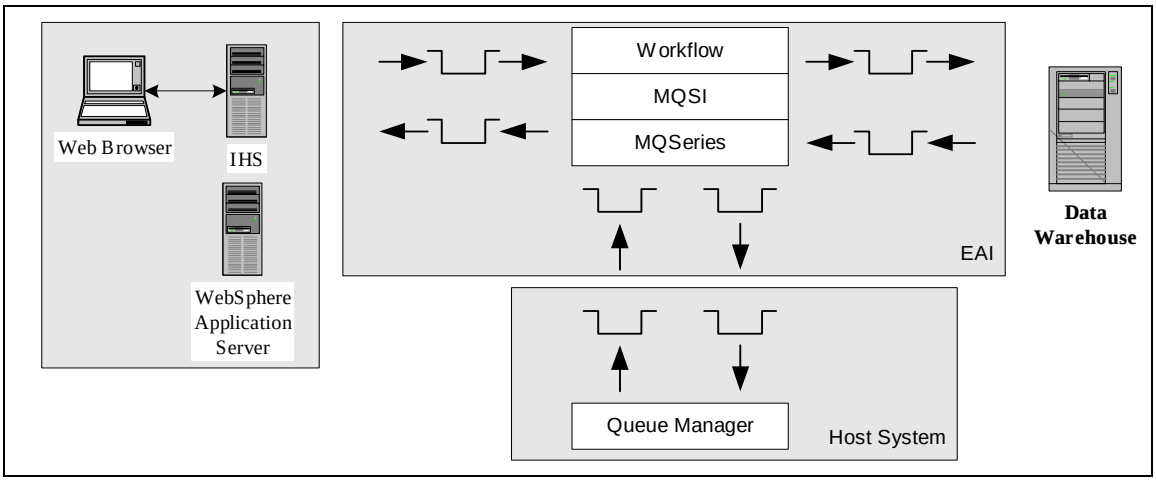

Figure 4 – EAI Routing of Data to Data Stores

3.1. EAI System Requirements

The EAI applications for the initial release of the Integrated Technical Architecture will require several hardware, software, and networking components that are outlined in the subsections which follow.

3.1.1. Servers and Workstations

EAI Servers

For the initial release of the Integrated Technical Architecture, two servers will be used to provide EAI functionality. One server will has been identified as the MQSeries Integrator server, the other will be used as the MQSeries Workflow server. MQSeries Messaging Queue managers for Sun Solaris will be installed on both servers. Each server will also serve as the hot-backup of the other.

EAI Workstations

EAI application development and monitoring will be performed using NT Workstations. These workstations will host the MQSeries Integrator (MQSI) Control Center and MQSeries Workflow Clients.

3.1.2. Networking and Interfaces

The EAI application will rely on SFA's local area network (LAN) environment for connectivity between workstations, printers, local servers and local hosts. The EAI application may also operate over the wide area network (WAN) that provides connectivity to legacy systems.

Local Area Network (LAN)

The network topology required for the EAI application is Ethernet. Transmission Control Protocol (TCP) is the communications protocol used between the workstations and the servers. The LAN connection will support the following:

- Connectivity to the EAI servers.
- Connectivity to the WebSphere Application Servers
- Connectivity to the Systems Network Architecture (SNA) server, if required.
- Connectivity to the WAN, if required.
- Connectivity to the network printer(s).

Wide Area Network (WAN)

The standard communication protocols that will be used by the EAI application over the WAN are SNA and TCP/Internet Protocol (IP). The WAN may also provide connectivity for the EAI to interface with existing SFA legacy systems.

Interfaces with external Systems

There are a number of interfaces that will be required to enable the EAI components within the production system. For the initial release of the Integrated Technical Architecture, this includes interfaces between the EAI application and the WebSphere Application Server, the Data Warehouse Environment, and Customer Information Control System (CICS) applications running on an OS/390 mainframe. The message flows between these systems will be accomplished using queues, which are part of the MQSeries Messaging layer of the EAI application.

The diagram depicts an example of the message flow between the application server, a legacy system, and the data warehouse.

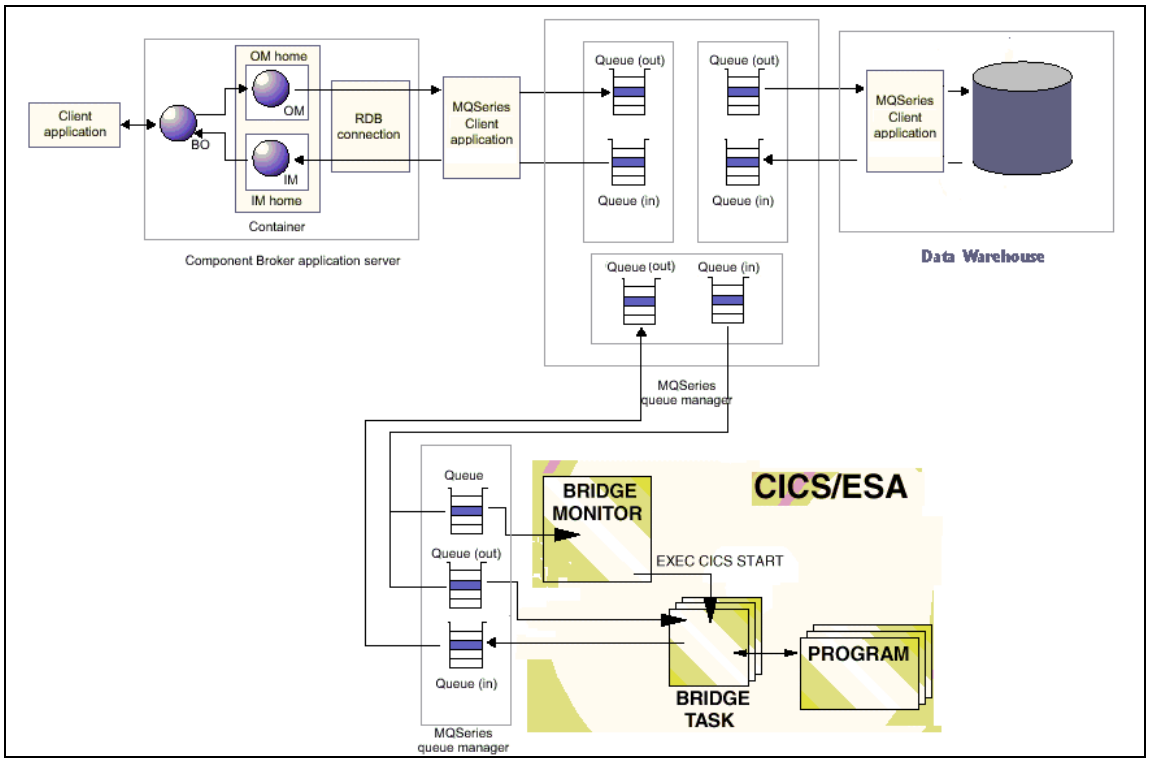

Figure 5 – Message Flow

3.2. EAI Development Environment

The EAI development environment consists of a single server running Sun Solaris and NT workstations for application developers.

MQSeries Integrator will be implemented in a two-tier architecture. The MQSI server components will be installed on a central development server while the Command Center must be installed on end-user NT workstations.

Implementation of MQSeries Workflow in the development environment will also be 2-tier. The MQSeries Workflow Server components and Buildtime as well as the Database server will be installed on the EAI development server. The MQSeries Workflow Client components will be installed on the NT workstation.

Both MQSeries Integrator and MQSeries Workflow require a DB2 UDB database to be installed on the development server. The DB2 UDB database serves as the metadata repository for both applications. The database comes bundled with both products.

3.2.1. EAI Development Server

Hardware

The hardware used for the Development environment is a Sun E3500 server. The hardware will be installed, configured and tested at the Virtual Data Center (VDC).

The minimum hardware requirements for the development server include:

- 400-MHz Sun Sparc processor
- 2.5GB Hard Drive
- 2GB RAM

Software

The EAI development server requires installation of the software listed below. The software will be installed, configured and tested at the VDC center.

- Sun Solaris version 2.6
- SunLink SNA for version 9.1 (if required to connect to the legacy systems)
- MQSeries Messaging for Sun Solaris version 5.1
- MQSeries Integrator (MQSI) Server components for Sun Solaris version 2.0
- MQSeries Workflow 3.2.1 Server Components and Buildtime for Sun Solaris or later
- DB2 UDB (bundled with MQSI version 2.0, and MQSeries Workflow)

3.2.2. EAI Development Workstation

Hardware

The minimum hardware requirements for the EAI development workstations include:

- 500-MHz Pentium III Processor
- 10GB Hard Drive
- 512K RAM
- 1024 x 768 video monitor

Software

An EAI development workstation requires the following software to be installed:

- Windows NT Server version 4.0 with Service Pack 5
- MQSeries Integrator Command Center
- MQSeries Workflow Client

- MQSeries Client
- DB2 Client

3.3. EAI Production Environment

For The initial release of the Integrated Technical Architecture, two servers running Sun Solaris and NT workstations will be used to provide EAI functionality in the production environment.

MQSeries Integrator will be implemented in a two-tier architecture. The MQSI server components will be installed on the MQSeries Integrator production server.

Implementation of MQSeries Workflow in the production environment will also be two-tier approach. The MQSeries Workflow Server components will be installed on the MQSeries Workflow server. The MQSeries Workflow Client components will be installed on the NT workstation.

Both MQSeries Integrator and MQSeries Workflow require a DB2 UDB database to be installed on their respective server. The DB2 UDB database serves as the metadata repository for both applications. The database comes bundled with both products.

MQSeries Messaging Queue managers for Sun Solaris will be installed on both servers. Each server will also serve as the hot-backup of the other.

3.3.1. EAI MQSI Production Server

Hardware

The hardware used for the MQSI production server is a Sun E3500. The hardware will be installed, configured and tested at the VDC.

The hardware configuration for the MQSI server is as follows:

- 400-MHz Sun Sparc processor
- 5GB Hard Drive
- 4GB RAM

Software

The MQSI Production server requires installation of the software listed below. The software will be installed, configured and tested at the VDC center.

- Sun Solaris version 2.6
- SunLink SNA for version 9.1 (if required to connect to the legacy systems)
- MQSeries Messaging for Sun Solaris version 5.1

- MQSeries Workflow 3.2.1 Server Components for Sun Solaris or later
- MQSeries Integrator (MQSI) Server components for Sun Solaris version 2.0
- DB2 UDB (bundled with MQSI version 2.0, and MQSeries Workflow (MQWF))

3.3.2. EAI Workflow Production Server

Hardware

The hardware used for the MQWF production server is a Sun E3500. The hardware will be installed, configured and tested at the VDC.

The hardware configuration for the MQWF server is as follows:

- 400-MHz Sparc processor
- 5GB Hard Drive
- 4GB RAM

Software

The EAI server requires the software listed below. The software will be installed, configured and tested at the VDC center.

- Sun Solaris version 2.6
- SunLink SNA for version 9.1 (if required to connect to the legacy systems)
- MQSeries Messaging for Sun Solaris version 5.1
- MQSeries Workflow 3.2.1 Server Components for Sun Solaris or later
- MQSeries Integrator (MQSI) Server components for Sun Solaris version 2.0
- DB2 UDB (bundled with MQSI version 2.0, and MQSeries Workflow)

3.3.3. EAI Workflow Workstation

Hardware

The minimum hardware requirements for the EAI production workstations include:

- 400-MHz Sparc processor
- 5GB Hard Drive
- 4GB RAM

Software

An EAI production workstation requires the following software to be installed:

- Windows NT Server version 4.0 with Service Pack 5
- MQSeries Workflow Client
- MQSeries Client
- DB2 Client

3.3.4. EAI Physical Interfaces

The EAI application will have the ability to interface with web-based applications, relational databases, COTS packages, and legacy systems.

Each EAI interface has specific requirements. Refer to the EAI Adapters and Bridges Overview (section 6) for additional details regarding these interfaces. The following sections outlines specific interfaces required for the initial release of the ITA.

EAI CICS Interfaces

MQSeries bridges will be used for accessing CICS to provide the benefits of a message / queuing model and to facilitate application-to-application communication.

There are several facilities available in CICS, which enable a (new) front- end application to interact with both MQSeries and with existing CICS transactions. Using a combination of FEPI, EPI, EXEC CICS LINK and EXEC CICS START, it will be possible to access the majority of existing CICS transactions.

- *FEPI and EPI* can be used to access existing transactions, which require a (3270) terminal oriented interface.
- *EXEC CICS LINK* can be used to access CICS programs (rather than transactions) which have an existing 'link' interface.
- *EXEC CICS START* can be used to invoke asynchronous transactions. Note that obtaining a response from these sorts of transactions may require a more extensive structure than the example below.

The EAI CICS interface requires a Queue Manager to be installed on the legacy system. The Queue Manager in the EAI MQSI production server will communicate with the Queue Manager on the legacy system.

The diagram below depicts the EAI CICS Interface.

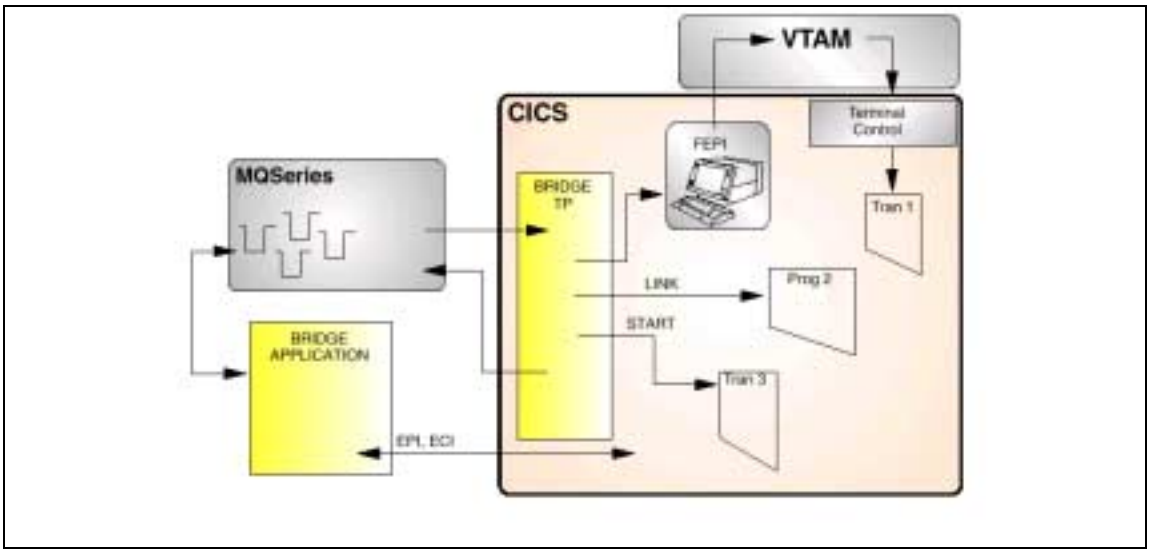

Figure 6 – EAI CICS Interface

CICS DPL

MQSeries provides a bridge for MVS/ ESA for CICS DPL and a bridge for CICS 3270 transactions. The diagram below depicts the use of the MQSeries CICS DPL. It is important to note that a MQSeries queue manager must be defined on the legacy system for this interfacing approach.

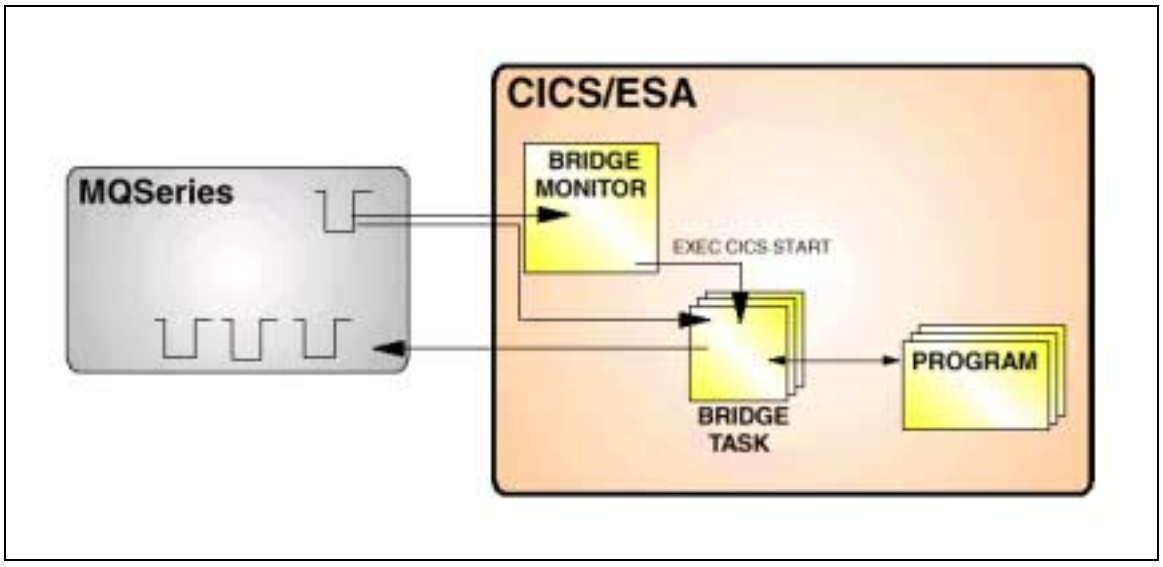

Figure 7 – MQ Series CICS DPL

CICS Transaction Server V1. 3

The diagram below depicts the use of the MQSeries CICS Transaction Server version 1.3 for CICS 3270 transactions. It is important to note that an MQSeries queue manager must be defined on the legacy system for the Transaction Server approach to work.

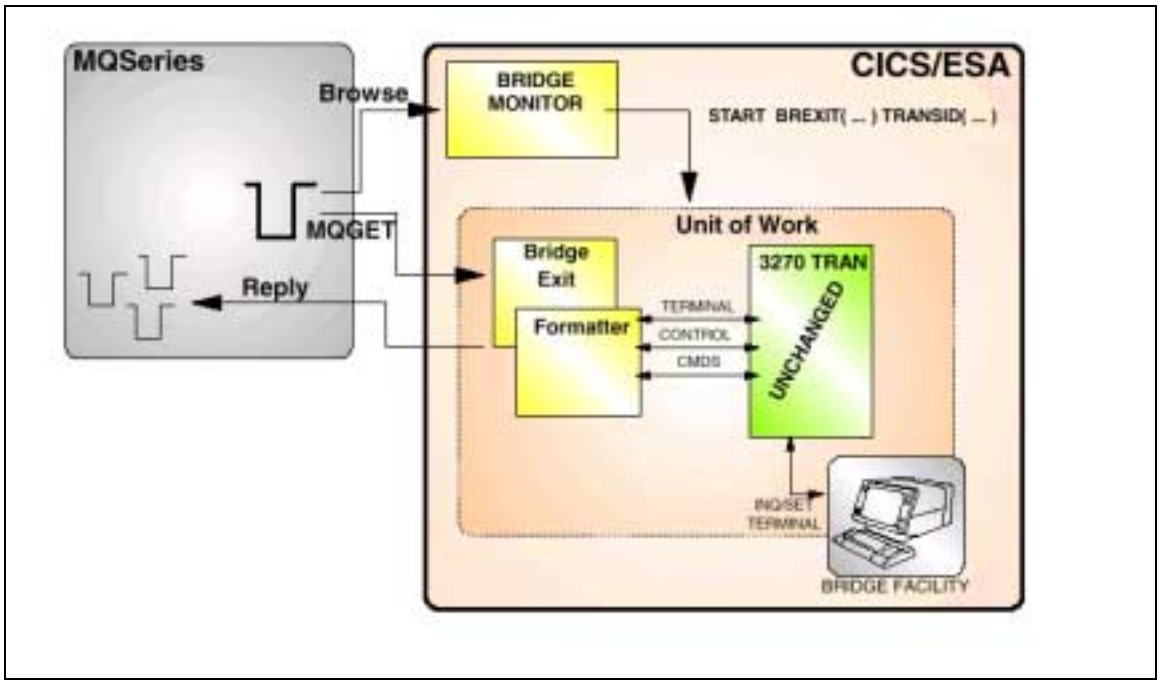

Figure 8 - MQSeries CICS Transaction Server

Note: The specific approach or approaches that will be used to interface between the EAI MQSI production server and existing CICS applications will be determined during the build and test phase of the Integrated Technical Architecture. The selected target application will need to be evaluated to determine the best method of building the interface.

EAI Application Server Interface

The EAI application will interface with the WebSphere Application Server using the MQSeries application adapter (MQAA). MQAA provides integration capabilities between web-based applications and message-based applications that leverage MQSeries messaging.

The WebSphere Application Server will implement an MQ client to connect to the queue manager on the EAI MQSI Production Server over TCP / IP via the MQ listener.

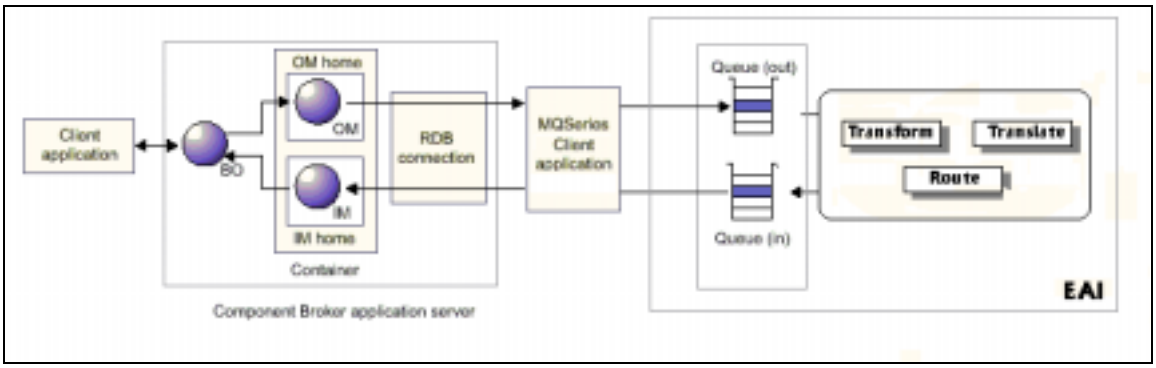

Figure 9 - Sample MQSeries Client Application Implementation to access the EAI application

The diagram above shows an example of an MQSeries client application implementation to access the EAI application. In the example, an MQSeries client application communicates with the EAI system via two MQSeries queues. These queues reside on the EAI production server. The access is via the MQSeries Application Adapter. The EAI application, using MQSI, transforms the data. The queue manager on the production server routes the data via MQSeries to the legacy system. Additional standards will be released in the future to allow the integration of the EJB and the EAI application to further enhance the communication between the Application Server and the EAI application.

EAI Database Interface

The diagram depicts the interface between the EAI MQSI Production Server and a database server. The database, such as a data warehouse is remote from the application and queue manager, which is acting as the coordinator. The database client libraries must be installed on the EAI MQSI Production server and can transparently connect to server processes on either the same or different physical machines. The interface between EAI and the database is through a custom application.

The custom application is an MQSeries client application that relies on the MQSeries Application Programming Interface (API) and the database API. Using the MQSeries API, the application can access MQSeries queues. Using the database API, the application, via embedded Structured Query Language (SQL) statement can access the database. Data will be placed in MQSeries defined queues.

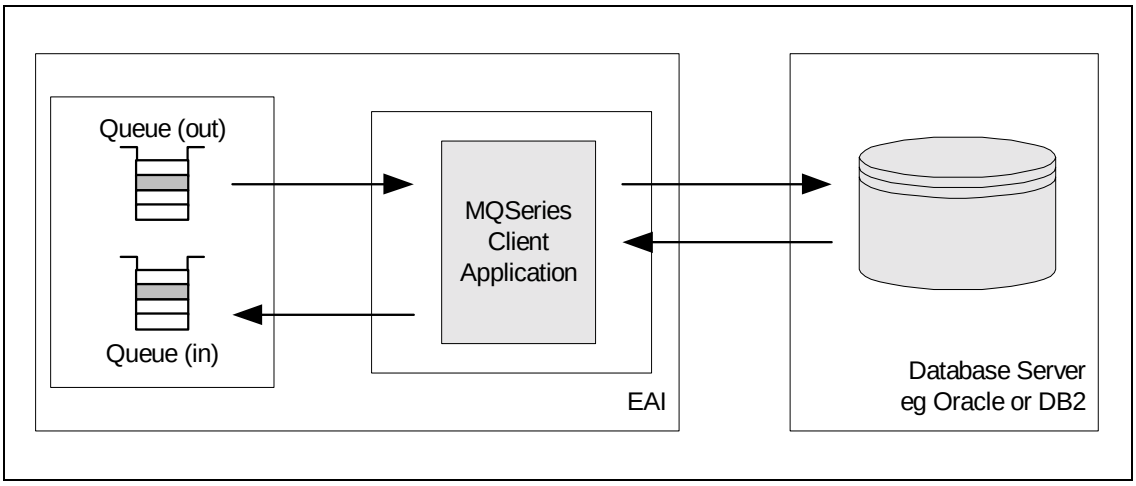

Figure 10 - EAI MQSI Production Server Database Server Interface

3.4. Operations Environment

The operation management of the EAI system will be performed using third party tools and the standard tools that are provided by the product. The operation of the EAI system may include:

- Startup and shutdown of the EAI application
- Troubleshooting
- Scheduled and unscheduled maintenance
- Upgrades
- System monitoring

3.4.1. EAI Management with QPasa!

The EAI management will be performed using the QPasa! product. QPasa! is a middleware management software that addresses all five critical areas of configuration, operations, problem-detection, performance, and analysis. QPasa! will be used to support the MQSeries Messaging, MQSeries Integrator and MQSeries Workflow.

3.4.2. EAI Operations Workstation

Hardware

The minimum hardware requirements for the EAI operations workstations include:

- 500-MHz Pentium Processor
- 10GB Hard Drive
- 128GB RAM

- 1024 X 768 video monitor
- 16MB video RAM

Software

An EAI operations workstation requires the following minimum software to be installed:

- Windows NT 4.0
- Exceed
- QPasa!
- MQSI Control Center
- MQSeries Workflow client

3.5. Future Implementation Options

In future releases, additional technical components may be required to provide additional scalability, flexibility and availability within the SFA technical environment.

The EAI application is designed to be redundant, highly available and scalable. However, the system design will be implemented in stages.

EAI Clustering

The EAI application is designed using the MQSeries products that provide a highly redundant and scalable system. In order to meet all the requirements for the EAI application, it is recommended that the EAI application get implemented using the clustering portion of the MQSeries Messaging system. The EAI cluster will consist of a minimum of two EAI physical servers, each running a full version of the EAI software as shown in the diagram below.

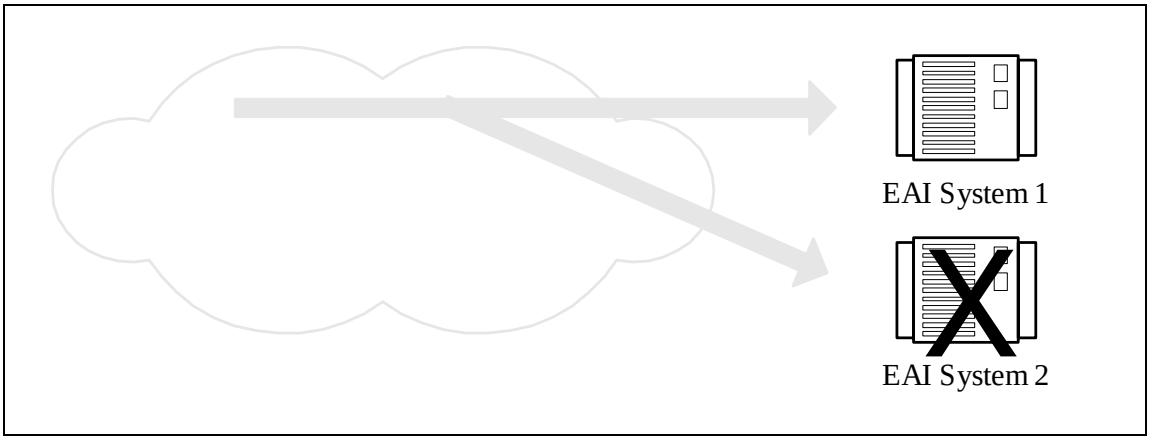

Figure 11 – EAI Cluster with Dual Physical Servers

In this configuration, messages are sent to both EAI System 1 and EAI System 2. Two MQSeries will handle these messages queue managers. This configuration improves the processing rate of the messages received. Note that only one message is sent to a server; it is not replicated two times, rather a specific sever is chosen and the message is sent there. Also note that placing and receiving the messages from the client sides (Internet space and legacy systems) is still local and totally transparent.

If EAI System 2, for example, becomes unavailable, then it is not sent any further messages and EAI application 1 will process all the messages.

It is important to remember that messages are owned by one queue manager rather than shared between the two queue managers, if it fails, messages are held (if persistent) until the queue manager restarts. New messages will not be sent to a failed Queue manager.

From this example, we can see how a cluster can provide a highly available and scalable EAI messaging system.

EAI Clustering Implementation

The implementation of the EAI application clustering may be performed in future releases of the Integrated Technical Architecture. For The initial release of the project, EAI clustering will not be implemented. Depending on budgets and other factors, EAI clustering can be implemented at later stage without a great impact on the overall system design.

Additional Interfaces

In future releases of the Integrated Technical Architecture, additional interfaces may be required to access information from systems outside the scope of The initial release. Some of these interfaces may include:

- RDBMS other than Oracle or DB2
- COTS packages (i.e., Siebel, Oracle Financials)
- Existing legacy systems

Refer to section 6 of this document for an overview of other adapter and interface methods.

4 Communications Middleware (MQSeries Messaging)

MQSeries messaging is centered on the idea of time-independent and connectionless communication of messages. A message is a collection of data; a string of bits and bytes that have meaning to a particular application, and formatted by that application to convey meaning in a fashion that it recognizes.

In the MQSeries Messaging environment, messages are exchanged between parts of a distributed application by placing them on a queue. An application can define any number of queues that it uses, and each queue can be used for a distinct purpose, or to convey a distinct type of message.

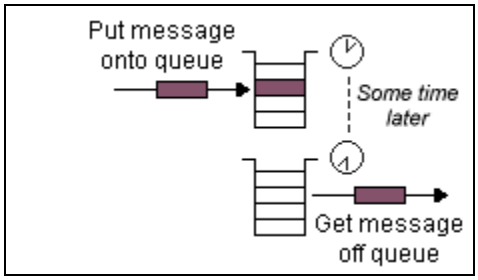

Figure 12 – MQ Series Messaging Environment

Generally, each queue is used to control the flow of information from one application to another application in one direction. One application puts messages on a queue and, at any time later, another application gets messages off the queue. When an application gets a message off a queue, the message is removed from the queue.

Two queues can be coupled to provide two-way communication between applications.

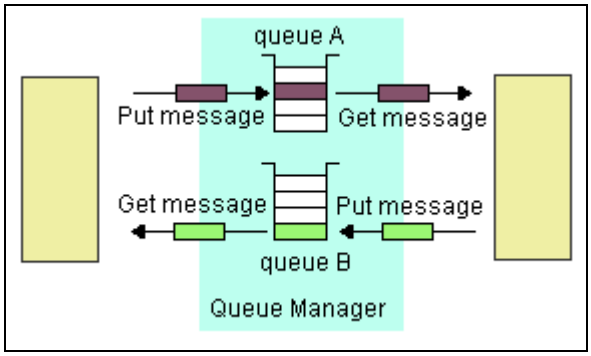

Figure 13 – Two-way Queue Application Communication

The queues defined on a given server are controlled by a local queue manager. More than one queue manager can be created on the same server. In fact, different applications may use

different queue managers to create, define, and control their queues. Queues can be created either administratively or programmatically by the applications that use them.

Queue managers on different servers can be interconnected administratively so that each present the queues they control as a local queue on their respective machines. For the applications that access a given queue, the queue will be local to each application. However, the queue managers interconnect these local queues so that they form a single logical queue. The queue manager transfers messages from one queue to the next to complete the circuit between the application that put the messages, and the application part that gets the messages.

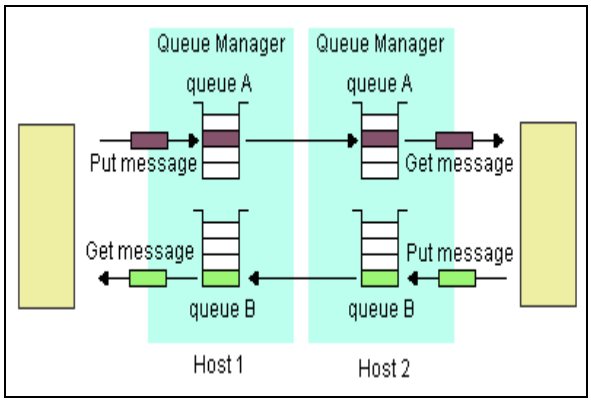

Figure 14 – Queue Manager Message Transfer

A fundamental characteristic of message-queue based communication is that it places the focus on the flow of information rather than the flow of control. Unlike call-based communication models, the "requesting" program is not blocked while work is being performed. Rather, the requesting program merely puts information onto a queue then continues its own work. Having sent its information, the program does not know or care where, when, or how that information is actually sent and processed. In this way, the functionality of the program that sends the information is completely independent of the program that receives and processes it. In fact, the sending program does not even know what the receiving program is, let alone what it does.

The receiving program could be unavailable at the time the information is sent; the queue manager ensures the information is not lost and gets transmitted when the receiving program becomes available. The receiving program could simply record the information captured in the message at one extreme, or perform some complex function on the information at the other extreme. It could even reformat the information and send it downstream on another queue.

To exploit the characteristics of queue-based messaging, a common design pattern for each part of the application is to simply remain in a loop; receiving information coming into an inbound queue, processing the information, then putting the results onto an outbound queue.

The overall flow of information between the application parts is important to the enterprise as this represents the effective processing of their enterprise business. But the disconnected nature supported by message queues allows different application developers to concentrate on their own application function independently of other programmers developing other parts of the application.

MQSeries supports a wide variety of *qualities of service*. These qualities can be defined administratively on the queue manager, on individual queues, or merely in the way a program interacts with the queue manager or queue.

The use of message queues can be arranged to achieve a number of overall communication models. These include unidirectional, bi-directional, client-server and parallel processing communication.

4.1. Unidirectional Communication

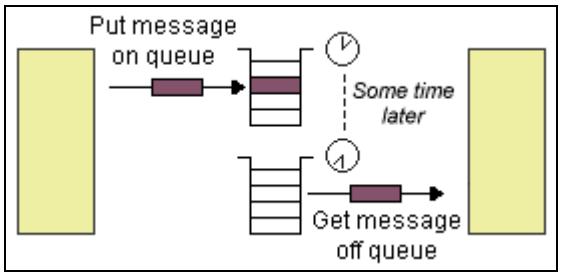

Figure 15 – Unidirectional Queue Application Communication

With Unidirectional communication, a single (logical) queue is created between two applications. The sending application only puts messages on the queue, and the receiving application only gets messages from the queue. Each application works completely independently of the other. In particular, there is no reverse communication from the receiving application to the sending application. The sending application is never aware of what the receiving application has done, or even when it has actually received its messages and processed them.

4.2. Bi-directional communication

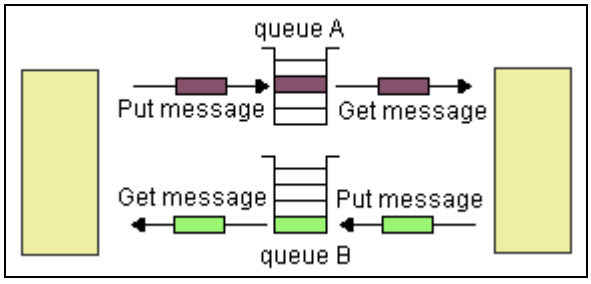

Figure 16 – Bidirectional Queue Application Communication

With bi-directional communication also referred to as request-reply messaging, two queues are created between the two applications. The sending application puts request messages on a request queue. The receiving application gets these messages and processes them. When processing is complete, the receiving application puts a reply message on a reply queue. The sending application can then get the messages from the reply queue and use the results in some other computation.

4.3. Client-server communication

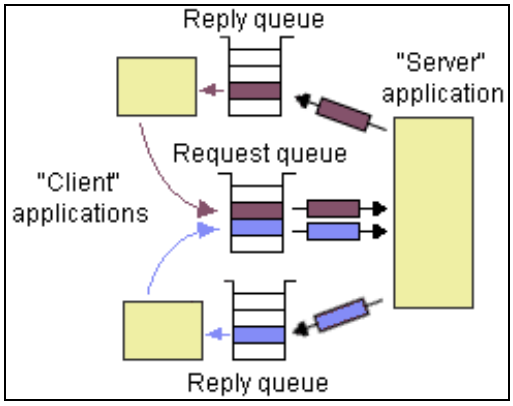

Figure 17 – Client Server Application Communication

Client-server communication builds on request-reply messaging to handle requests from one or more clients. One application is designated as the server. One or more other applications are designated as clients. Each client shares a common request queue, but each has a separate reply queue. The server gets messages off the request queue and can put messages onto all of the reply queues. After processing a request message, it puts the results in a reply message on the reply queue for the specific client that issued the request.

4.4. Parallel processing communication

Basically reverses the client-server communication style. In parallel processing, one application is designated as the task manager. One or more other applications are designated as subtasks. A separate request queue is created for each subtask. All subtasks share a common reply queue with the task manager. The task manager divides up the work and places a distinct message on the request queue for each subtask. Each subtask gets messages from their respective request queue. After processing each request, each subtask puts the results in a reply message on the common reply queue.

4.5. Queue Design Recommendations

Each messaging architecture above is a valid approach for implementation in the Integrated Technical Architecture Environment. However, the appropriate method must be evaluated at the application level on a case-by-case basis. For example, unidirection communications

messaging may be valid for certain applications that do not require acknowledgement that the receiving application has, in fact, received and processed a particular message. Each business capability that will use the Integrated Technical Architecture will need to determine the appropriate messaging architecture for their particular applications.

5 Transformation and Formatting (MQSeries Integrator)

MQSeries Integrator Version 2.0 is the product within the MQSeries family that provides transformation and formatting capabilities within the Integrated Technical Architecture. It is a framework designed to enable the provision of mission-critical Business Integration tools and processes. It helps build new solutions and enhance existing solutions by adding functionality to existing programs and data without requiring changes to the existing programs and data.

This overview is designed to provide a concise look at the technical architecture and functionality of MQSeries Integrator . The overview will cover the following:

- MQSeries Integrator Overview
- Message Flow Framework and Architecture
- Message Dictionaries
- Message Warehousing
- Publish and Subscribe
- Multi-Broker Domains
- Connecting Applications to the Broker
- Internal Architecture and Administration
- Broker Tools and the Control Center
- System Management
- Security
- Migration Issues
- Summary

5.1. MQSeries Integrator Overview

MQSeries Integrator provides a simple yet sophisticated way to process messages en-route to their destinations. This will allow SFA focus on how to use this enabling technology to transform their business processes, without needing to change their existing applications.

MQSeries Integrator functions as a *Message Broker*, which provides both the MQSeries messaging layer and the Message Brokering hub for processing, transformation and distribution of messages. Potential business uses for MQSeries Integrator include:

- Efficient connectivity for application integration using hub and spoke model
- Transformation of data while routing between applications

- Separation of business logic from application logic and data logic
- Added business application functionality such as publish / subscribe
- Integration framework for adding existing and new vendor products to further add value
- Seamless integration of messages and relational databases
- Mapping between XML message formats and other data formats

Message Brokers act as a hub for messages passing between MQSeries applications. Once messages have reached the Message Broker, they will be processed, based upon the configuration of the Message Broker and on the contents of the message. Within the Message Broker, individual functions are assigned to a collection of interconnected *Node*s, where the processing and transformation activities take place.

A key component of MQSeries Integraton is enabling vendors and SFA developers to write their own processing nodes. Other components include an extended publish / subscribe facility, message dictionaries and message warehousing. These will be discussed in more detail below.

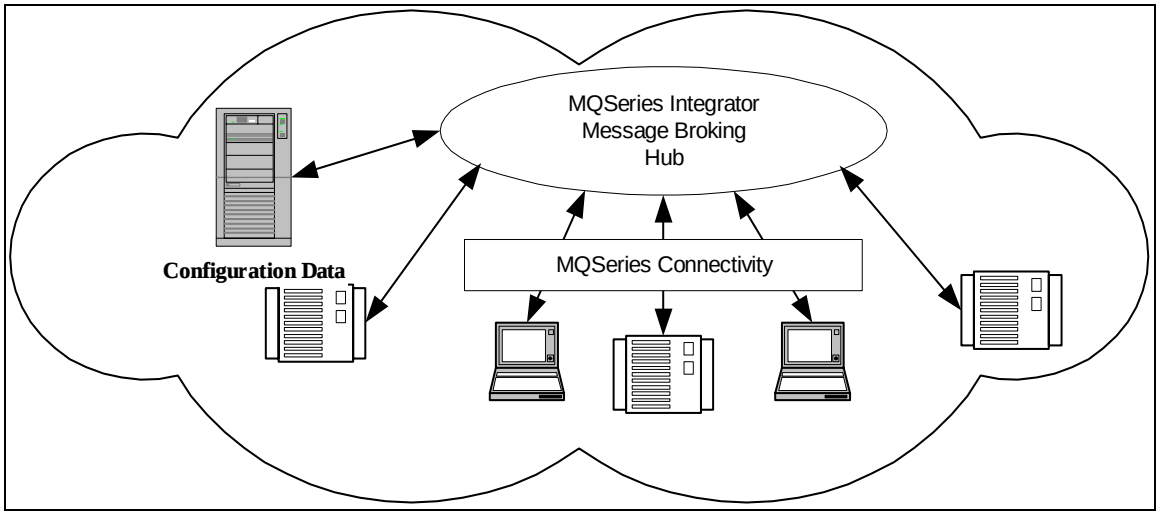

Figure 18 – MQSerices Integrator Message Broking Hub

As the diagram shows, the Transport layer for the progression of messages into and out of the Message Broker is the MQSeries Messaging product. Many of the functions of MQSeries Messaging, such as assured once-only delivery and transactional support for messages are leveraged within the MQSeries Integrator application.

5.2. Nodes

Within the framework of a Message Broker, all processing on the messages is performed within constructs called Nodes, which are called from the message broker's execution

environment. These *Message Processing Nodes* perform operations on the message data within the message flow, as well as having the ability to access information outside the message flow, such as accessing a database. Nodes may have Input and Output *Terminal*s, and connectors logically join up the terminals. Through the design and configuration of a selection of nodes, a sophisticated processing environment can be built.

Nodes have a set of terminals that are used to receive messages from connectors or send messages to connectors. Common types of terminals are *in* terminals, which receive messages, *out* terminals, which forward on messages, and *failure* terminals, which forward on messages when some error has occurred during the processing of the message or an exception has been raised. This allows an EAI application developer to specify the behavior of the system when encountering a failure at a particular point in the message flow.

A message processing node is a well defined processing stage, coded to perform a specific task, or set of tasks on a message flowing through the broker. A selection of predefined nodes comes supplied with MQSeries Integrator. Additional nodes can be created and plugged-in. In order to form a message flow within an instance of a Message Broker, any number of nodes can be 'wired together' using connectors.

5.3. Message Processing Nodes

Processing Nodes perform the various operations on messages in the message flow. An input node initiates a message flow. As described below, MQSeries Integrator is supplied with an MQInput node that reads a message off a specified MQSeries Queue. This node will be connected to other nodes.

Although the links between nodes are called connectors, these are purely constructs to assist in wiring the nodes together in the Control Center graphical tool. Messages are actually passed between nodes by method invocation calls with a pointer to the message object passed between the nodes. The properties of the nodes in each message flow can be customized. This will enable the function of the nodes to be specific to the messages flowing through the nodes, and to enable the processing required of the nodes in the message flow to be performed.

For example, in a node that will perform a filter operation, the filter statement is assigned by customizing the node specifically for the appropriate message and the filter operation that will take place. In an MQInput node the name of the associated MQSeries queue is given as part of the customized information along with the transactional properties of the message flow.

5.4. Transactionality and Threading Support

Processing Nodes do not explicitly maintain the transactional integrity of the messages flowing through the system. Instead the integrity is maintained transactionally within the bounds of the message flow. A message is read from an MQSeries application queue to begin the message flow and placed onto an MQSeries application queue to terminate the

flow of the message through the instance of the Message Broker. In between the MQInput node, which begins the message flow, and the termination of the route through the message flow, the message follows the processing route through the nodes in the message flow. If there is more than one connection to an output terminal for a node, and the processing is such that the message will be fired through this terminal, the processing for each different route through the subsequent nodes attached to those output terminals will be handled independently and sequentially. This will maintain use of the same thread within that message flow, until the message flow for that message is complete. At that point the thread is released back into the thread pool.

There can be many instances of the same message flow all processing messages read from the same MQSeries queue. The system is multithreaded and designed to allow many messages to be processed concurrently in multiple instances of a message flow, thus providing multiple instances of each node. It is always in the interests of the application developer designing the messages to flow through the node, and the node developer (if a user-developed node), to ensure that the messages can flow rapidly through nodes. If messages perform lots of different tasks within the message flow, and can become I/O bound, then the performance of the message flow in terms of throughput of messages will be adversely affected.

Each message flow is allocated a pool of between 1 and 256 threads. Each message that comes into a message flow is assigned to a separate thread in the following manner. For each MQInput node in a message flow there is a listening thread. This thread waits for a message to be placed onto the input queue specified to that node. The thread then actions the message, while a second thread is started to take on the role of listening for incoming messages in another message flow instance.

When the message being processed by the original thread completes its processing within the message broker, the original thread is returned to the thread pool for subsequent usage. However, should all threads in the thread pool be used, the next thread to be returned to the thread pool will be used to monitor the input queue.

If a message flow is to be transactional and to include a database operation in the message flow, then the MQInput Node and a Database Node will need to be configured to accept coordinated transactions.

5.5. Supplied Node Description and Functionality

Up to this point only processing nodes have been discussed. However, there are a large assortment of pre-defined nodes that are part of the MQSeries Integrator product. Each node is briefly outlined below, with a description of the function attributed to each type of node. All nodes described below have an in terminal, and a failure terminal. Some nodes have a variety of output terminals that vary depending on the type of node. Exceptions will be noted for each node described.

5.5.1. Triggering and Initiation

The only basic node type that offers this is the *MQInput* Node. The MQInput Node uses an MQGET call to receive a message from a queue. The message then proceeds onwards in the message flow either through the out terminal of the MQInput Node, on to the subsequent node, or to the failure terminal, if an error occurs. There is a third terminal, called catch. This terminal is initiated if the exceptions occur later in the message flow and are not handled closer to the point of failure. One MQInput Node should be used to take messages off a single MQSeries queue. Problems relating to the sequencing of messages could arise if multiple MQInput Nodes were reading from one MQSeries queue and message sequencing is not used.

5.5.2. Checking and Filtering

The basic node types here are the Check Node and the Filter Node.

- The *Check node* checks whether the message's '*Message Type Specification*' matches the expected attributes for some or all of the attributes *domai*n, *set* and *typ*e. This enables evaluation of MQSeries Messaging message headers and other standard properties. Messages flow through the in terminal. Should the check be successful they are flowed through the match terminal; otherwise, they are routed through the failure terminal.
- The *Filter node* is a content-based evaluation of the input message, using an SQL expression as the decision criteria. The possible terminals for this node are *in*, *tru*e, *fals*e, *unknown* and *failure*. The message is flowed to the *unknown* terminal if the result of the evaluation is indeterminate or unknown. If a failure occurs (such as an arithmetic overflow) during the evaluation, the message is routed through the *failure* terminal.

5.5.3. Message Manipulation

The basic node types here are the Compute Node, the Extract Node, the NEON FormatterNode and the ResetContentDescriptor Node.

- The *Compute node* is designed to transform a messages, with the ability to accept one message type as an input and after transformation to output a different message type. This is done using the contents of the input message and optionally data values from an external relational database. Each element of the output message, which can be entirely different from the input message, can be derived using a specific formula, with the language used to specify the query closely based on SQL3.
- The *Extract Node* will produce a transformed output message from the input message by selecting, copying, and modifying data elements from the input message to create a new output message.
- The *NEON Formatter* Node invokes the NEON Formatter engine to map the content of the input message to the output message. By using this node a message in one format, defined to the NEON Repository can be transformed into another message format as

defined in the NEON format definitions. This node can be used either on its own or in combination with other nodes to build up a comprehensive message flow.

- The *ResetContentDescriptor Node* is designed to allow a message to be interpreted by another parser type to within the same message flow. It performs the same function as if the message was passed from an MQOutput Node to an MQInput Node.

5.5.4. External Database Operation

The Basic Node types here are DataInsert Node, DataUpdate Node, DataDelete Node, Database Node and Warehouse Node. All these Nodes are specialized nodes that perform a specific function in accessing a particular database. All of these Node Types have *in*, *out*, and *failure* terminals. All the operations using these nodes can be part of an externally coordinated transaction, or they can commit the transaction independently. None of these nodes alter the message the flows through them. The input for the DataInsert, DataUpdate and DataDelete nodes must be typed, but the input for the Database node can be generic XML.

- The *DataInsert node* performs a single insert of a new row into a specified database. Some of the information from the message may be used as part of the insert or the message may just be used as a trigger, possibly following a filter node. An internally generated SQL statement is used to do the insert.
- The *DataUpdate node* will update the values in one or more rows of a specified database. An internally generated SQL expression is used for the update.
- The *DataDelete node* can delete one or more rows from a table in a specified database. The message is unchanged in the process. Data from the input message can be used in the expression to specify what data is deleted from the database.
- The *Database node* can perform a database operation on a specified database, without changing the message, which passes through from the in terminal to the out terminal. Values from the message can be included in the SQL expression to execute the database operation.
- The *Warehouse node*, which is similar to the DataInsert Node, is used to store the messages flowing through the message broker in a Message Warehouse. The messages stored in the warehouse may be stored there for audit purposes or for off-line or batch processing of messages or for subsequent retrieval and processing by the message broker. The message is added to the database in the warehouse using a SQL insert. The message is stored in the database with an index record built from the message schema.

5.5.5. Decision and Routing

The basic supplied node types are the MQOutput Node, the MQReply Node, the NEONRules Node, and the Publication Node.

- The *MQOutput node* is a defined endpoint of a message flow within the Message Broker. On leaving this node, messages are written to a MQSeries Queue using a MQPUT MQI

call. They can either be written to a specified fixed queue, or sent to a reply queue, or a list of destination queues can be specified, using information from the message.

- The *MQReply node* is a specialized version of the MQOutput Node. It is used when the MQSeries queue that a message is to be output to is the one specified by the ReplyTo field of the message header.
- The *NEONRules node* is used when the message needs to be passed to the NEONRules engine for evaluation of rules.
- The *Publication node* is used in the transmission of messages to subscribers of the defined publish and subscribe service. Messages routed through this node as part of a publish / subscribe message flow are matched to subscribers for both topic and content and then forwarded directly to local subscribers or routed to other brokers to match their subscribers that are remote to the publishing broker. Management of the published topics and the subscribers are handled elsewhere in the Control Center.

The diagram illustrates the integration of MQSeries Queues with the nodes in a message flow.

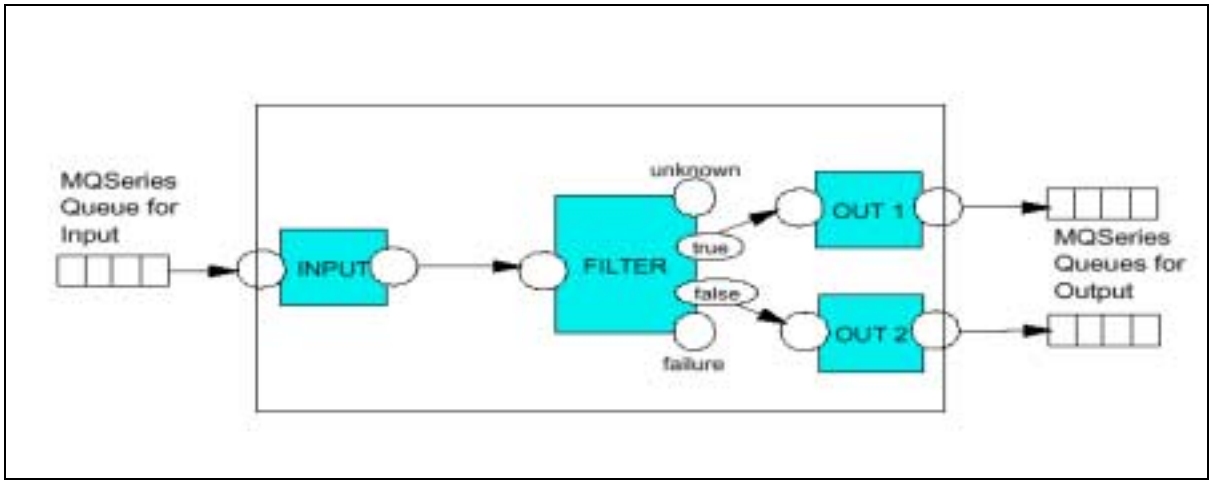

Figure 19 – MQSeries Queue Node Integration

Error Handling and Trace

There are three basic Node Types for error handling and audit trails. They are the Throw Node, the Trace Node and the TryCatch Node.

- The *Throw node* has just an in terminal and is used within the message flow to throw exceptions. These may be caught by TryCatch nodes earlier in the message flow or may cause the processing of that particular message to cease and associated transactional activity to be rolled back. Throw nodes may be used to throw an exception based on message content to prevent additional failures downstream in the message flow.
- The *Trace Node* is used to aid in debugging the message flow. It has an in terminal and an out terminal. The out terminal passes on the input message unmodified. However, the

Trace Node will format and write out a trace record to a specified destination, assisting in creating a record of the route a message has taken through the message flow. The trace format can be selected using a variety of options.

- A *TryCatch node* is used to prevent exceptions of downstream nodes from terminating the processing of messages or transactions, which is likely to happen if the exception drops back to the MQInput Node, which is the root node in the message flow. The message is received by an in terminal and forwarded on unchanged using the try terminal. If this node catches the exception it will be propagated using the catch terminal if this is connected. Error handling of the exception can then occur. If exceptions occur in the message flow have not been caught by other nodes, such as a TryCatch Node, the MQInput Node will catch the exception, as it was the initiator of the thread and the catch terminal on the MQInput node will propagate the message.

5.5.6. 3rd Party or Plug-In Message Processing Nodes

The Nodes described above are those which are supplied with the MQSeries Integrator product. 3rd parties have and will continue to develop additional Processing Nodes to enhance the message flow within the message brokers. In the future IBM may provide additional nodes as well. These nodes must be designed to match the requirements of the Message flow Framework that will allow the new nodes to be added to the MQSeries Integrator Design Tool.

5.5.7. Transactionality within the Message Broker

MQSeries functionality allows MQSeries messages to participate in coordinated transactions, thus ensuring the transactionality of the message within the system. By using this facility, messages within the Message Broker can also gain transactionality for their message flow. If there is any failure between doing the MQGET to read the message from the queue and doing the MQPUT to place the message and the failure or the exception is not handled, the entire operation can be rolled back to the MQGET.

Setting attributes on the instance of the message flow controls the flow of transactional behavior between the nodes. In the MQInput node there is a transaction attribute that determines whether messages flowing through the message flow will be handled transactionally. In the MQOutput node there is a persistence attribute to determine the persistence state of the outgoing message. The Database nodes have a co-ordination attribute to determine whether access to the database will participate in the transaction.

These attributes can specify factors such as whether information written to a database will participate within a transaction, how and when a transaction will be committed or rolled back, and whether the message within the message flow will be sent persistently between nodes. The EAI application developer needs to ensure that the appropriate level of transactionality is applied to each node and collection of nodes and to ensure that all participants, such as a database are appropriately configured. By using an appropriate combination of these attributes on the message flow, and the relevant nodes, a message flow

can be transactional for its entire span, or can be transactional only in accessing databases, or even be non-transactional.

5.5.8. Message Dictionaries

A key function of the MQSeries Integrator product is the ability to parse the contents of messages to either perform work on the message data or to allow the message data to drive work externally. In order to efficiently parse this data it is essential to be able to look up message formats in order to identify the relevant fields in each message for every node based function. Message Dictionaries are used to provide format information giving the ability to rapidly parse information from messages, held in the dictionary as a logical message model for direct access to named fields in the message body. With a message dictionary format provided to give templates for expected message types, the required element fields can be extracted from messages rapidly.

Components and Functionality of a Message Dictionary

A Message Dictionary provides support to parse the different formats of message contents and their associated headers that are defined to the Message Dictionary. These formats can include *MQMD* message descriptors, MQSeries *RFH*and *RFH2* format headers, *XML* messages, and messages built according to NEONFormatter definitions. These formats held within the dictionary can be collectively known as *Message Type Definition*s. Within the Message Dictionary, these definitions can be grouped together as *Message Set*s, and the Message Sets are deployed to the Brokers that will be processing the messages within these Message Sets.

The three main components of a Message Dictionary are a *Message Repository Manager (MR*M), a *Resource Manage*r, and a *Message Translation Interface (MT*I), but to a User the Message Dictionary is accessed from the Control Center user interface and the components are not exposed. The definition for the format of the messages, with identification of the fields and elements within a model message template, is known as a *Message Mode*l. The MRM uses the MQSeries Integrator Control Center tool to define and maintain the Message Model and stores its information in the MRM Database. The Message Models in the MRM can handle many forms of messages such as XML message formats and byte-oriented record structures from C or COBOL sources. This information is supplied to the Brokers to be held in a *Runtime Dictionary (RT*D), locally available to each deployed broker. The RTDs enable the Message Brokers to maintain a local cache of the format definitions and thus improve the speed of parsing the message formats. If the information in the dictionary is to be changed while a copy is held in the RTD, a new version of the dictionary is added, as dictionaries can't be modified once deployed.

Uses of the Message Dictionary

The Control Center is used as the definition tool for defining the message formats the Broker is expecting to receive. Messages to be received by the Message Broker have their formats defined within the Control Center, which is the interface of the Message Dictionary. These

defined formats are then used in conjunction with the processing nodes and parsers to provide the logical message formats used by the Message Broker from the wire format messages received via MQSeries. The messages are interpreted from the wire format descriptor with the aid of the message definition. The format of the message, when being defined using the Control Center, needs more than the types, the elements (or fields), and the associated lengths of the message in order to match the logical message format to the wire format.

The Message Broker does not just use the Message Dictionary for providing the logical message format from the wire format. The Message Dictionary is also used in the reverse way to take the logical message formats received by the MQOutput and Publish Nodes and to create wire format messages from those logical message formats. An example of this is where a COBOL program generates the content of a message and the content of the message is a COBOL record structure. The logical message structure held within the COBOL record can be defined by the Message Broker. This definition is then used to deconstruct the message into the relevant fields. However, when processing is complete within the Broker, and the message needs to be sent to another COBOL application as a message, the outgoing message needs to be created in wire format with the logical format becoming a COBOL record structure again. Once again the Message Dictionary is used to map the fields of the message, but this time it maps the message fields in those of the COBOL record, creating the wire format as defined in the Message Dictionary. One of the strengths of the logical format is that if the message is not being sent on to a COBOL application, but instead is being displayed on a Web Page and needs to be sent in HTML, the definition within the Message Dictionary will build the output wire format as required, taken from the definition in the Message Template.

Message Templates

The definition of each type of message, or related group of messages, is described as a Message Set. These Message Sets are assigned into a Message Dictionary, with a separate Dictionary for each Set. On receiving a Message, the information in the message header will help to identify the correct dictionary to load in order to parse the message. For deployment to brokers, the entire collection of related messages grouped together into a Message Set is assigned to the broker or brokers that will be receiving the messages for processing in the assigned Message Flows. The information within the message header that provides these details, and is defined to the Message Broker using the Control Center is as follows:

- The message domain, which describes the source of the message definition: that is, whether it has been defined by the Control Center tool or the NEON tool.
- The message set, or project, groups together a collection of messages, elements and types, within the specified domain, going to make up a complete definition of messages relating to a particular flow or business operation.
- The message type that will precisely define the structure of the data within the message, giving such details as the number and location of character strings.

- The message format that identifies the wire format of the message. These definitions will be required for each type of message (other than self defining formats such as XML), expected to be received by any message flow. When the definitions have been completed for each message, with the message set defined to a Message Dictionary, and the appropriate Dictionaries assigned to the message flow, then when a message is received, its type is identified by the information in the message header, and the appropriate Message Dictionary is accessed and finally the Message Parser is called to deconstruct the message. Careful thought must go into Message definition with the MRM. If messages are just being routed through the Broker with no transformation then only one message needs to be defined, as the output message is no different from the input message. If there is any transformation, with elements being added or removed from the message, then both input and output message formats should be defined to the MRM. When both input and output message formats have been defined, then when transformation of the message takes place, perhaps in a compute node, then the compute node will take the input format as the inbound message and the message will be transformed into the output message format according to the customized properties of the node performing the transformation. Note that the MRM does not distinguish between input and output message definitions. Whatever the sequence of messages used in the message flow, all messages are defined in an identical manner.

Message Parsers

Once the template has been defined the Parser is required to establish what is to be done to the received message. The Parser can be one that is supplied with MQSI, or it can be a user or 3rd party supplied Parser. The behavior and the Parser can be different, depending on whether the message has been defined to the Message Broker using the Control Center or not. New messages defined in the Control Center as parts of message sets are created to be logical message structures that can then be used in transmission to other systems. Supplied parsers for these messages will work on messages that use the following types of message headers: MQMD, MQRFH, MQRFH2, MQCIH (CICS bridge header), MQIIH (IMS bridge header). When the incoming message has been defined in the Control Center, but is generated by an application as a data structure, the parser can be specially constructed to deal with this or the data structure can be imported using the facilities of the Control Center.

Validation of message content is very useful when dealing with messages that have been defined to the Message Dictionary with attributes called Valid Values assigned to the message fields. Should the message not be defined using the Control Center, there are a number of options. The message could be in Generic XML, in which case it is self-defining and does not require a Dictionary definition. Should this be the case, the message cannot be validated to ensure it is in the correct format. If the message structure is not defined anywhere, it is treated as Opaque content and no content based processing can be performed on the message as it flows through the Broker, but it can be acted on according to the information in the Message Header and the design of the Message Flow.

5.5.9. XML and the Message Dictionary

If no template, or schema, exists for an XML message, but the message uses well-formed XML, the message can be termed self-describing and content based routing, based on the XML descriptors, can be applied. In this case, with a well-formed XML message, the message flow can still be architected, and nodes, such as the compute node can be customized to handle the elements in the XML message, even when they haven't been defined to the Message Dictionary. By using the message dictionaries for non-XML messages, which are not self-describing, allow content based routing based on the message dictionary's description of the content, once the message format has been defined to the Message Dictionary.

Within MQSeries Integrator, all configuration data for the application is held in XML format. Once a message is defined to the MRM as a message format, the format of the message to be output can be also defined with the data format for the output message being XML as required. Thus XML messages can be generated within the Message Broker from non-XML based message formats.

5.5.10. Message Warehouses

Message Warehouses can be used for many different purposes. However, a fundamental use is that they store long-lived messages in a standard relational database. These messages may then be used for purposes such as audit trails for a particular message type (such as high value messages). Message Warehousing may be a standard method of logging the work being performed within a particular broker, possibly for off-line analysis. Another possible reason for Data Warehousing would be to implement a full Data Mining or Data Analysis on the data flowing through the Message Broker.

Warehousing Nodes

In order to place a message within a Message Warehouse, a node must be configured to do so. This message flow is likely to perform some calculation or transformation on the message before moving the message to the Warehouse. The Warehouse Node may parse the message, or parts of the message that are to be moved to the Warehouse. The required format of the message and its data contents are created and passed into a built SQL INSERT statement for entry into the database.

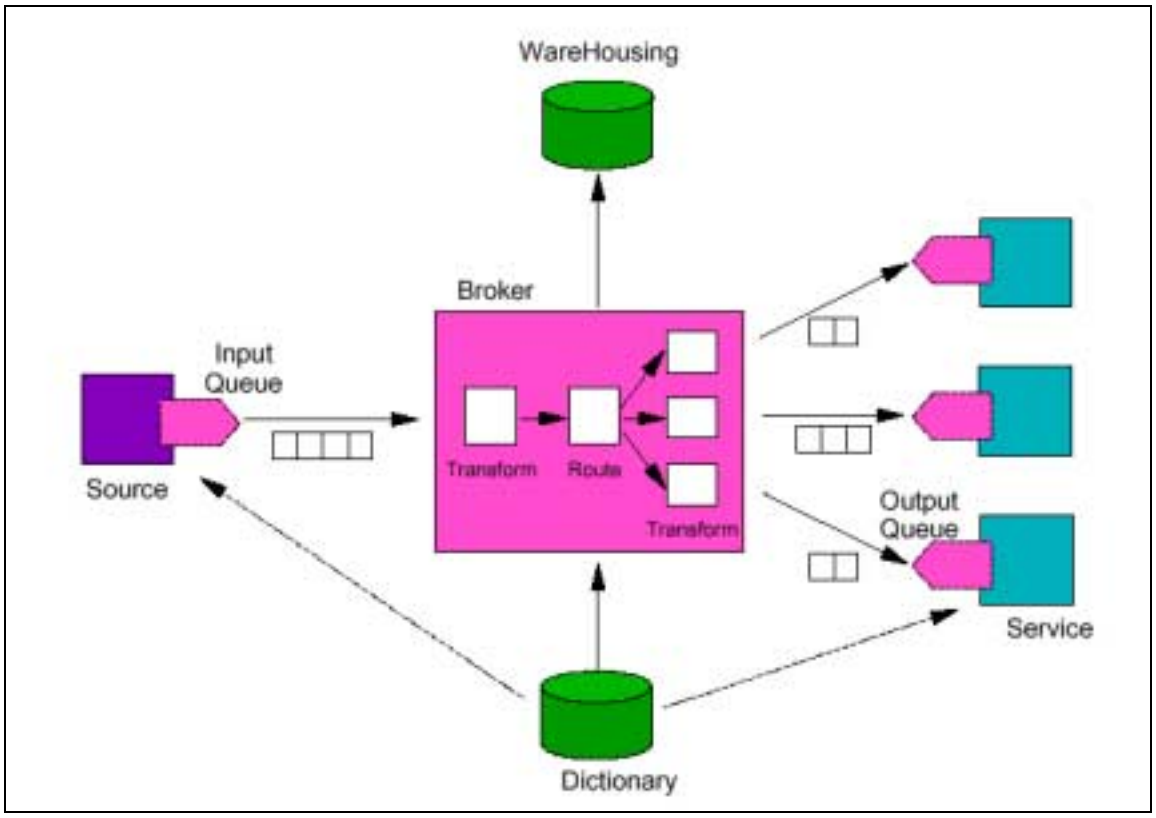

Figure 20 – Warehouse Nodes

5.5.11. Publish Subscribe System

Content and Topic based filtering

MQSeries Integrator provides content-based filtering on subscription, as well as the hierarchical topic based filtering that is available on both publication and subscription. For content-based filtering, the contents of elements within a message are evaluated by SQL expressions to establish the content filtering result. The content filters can be stored in the Dynamic Subscription Table. These filtered messages, when combined with the other supplied or defined Message Broker functions can be transformed for different applications and only the required parts of messages sent to the applications that subscribe to them. The publish / subscribe function is represented as a Node, known as a Publication Node within the message flow. The final action of a publish / subscribe node is the placing of a message on one or more MQSeries queues.

Subscription Handling

MQSeries Integrator supports two types of subscription handling. If subscribing applications for published data are known in advance of the publication, this is known as *static subscription* and routing can be well defined in advance. Subscribing applications adding subscriptions can change if the subscription set or changing subscriptions, this is referred to

as *dynamic subscriptio*n, which is flexible in terms of changing business requests within a system during runtime with no need to pre-register interest in particular message types.

The requests of subscribers to the publish / subscribe function take the form of messages known as *control message*s. These give the subscriber full ability to create, delete and change their subscriptions. The names for these messages are: *Register Subscrib*er, *Deregister Subscriber* and *Request Updat*e. For publishers there are different messages to meet their different needs. These message names are: *Publish* and *Delete Publicatio*n. These control messages are only required if the client application is using the MQI programming interface and not the MQSeries Application Messaging Interface or the MQSeries Java Messaging Service, as the MQI will need to build the headers explicitly to call these functions.

The diagram illustrates of the flows involved in the handling of dynamic publication and subscription requests.

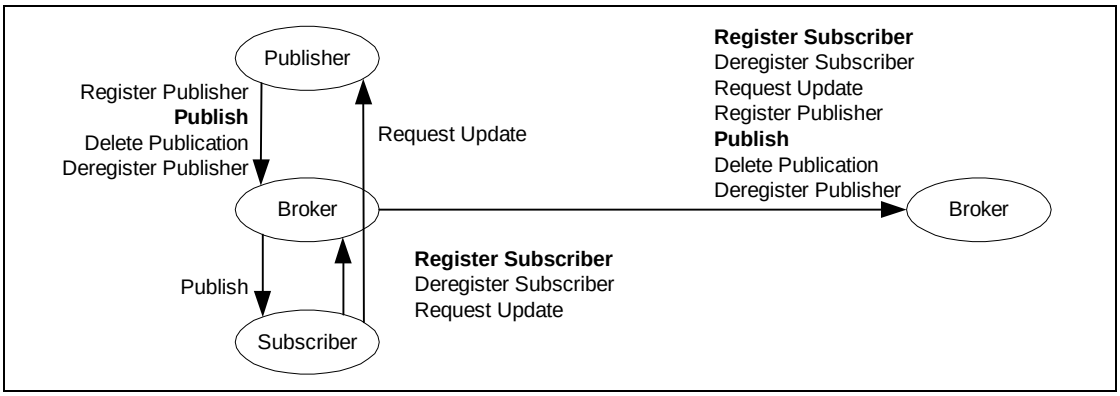

Figure 21 – Message Broker: Dynamic Publication

The list of subscriptions is held persistently within the Message Broker. Changes made to the subscription list are updated dynamically, and will take effect as soon as the message reaches the broker. The key items in determining publish / subscribe actions are as follows: topic, content, subscription point and destination. Any combination of these four items can be used to create a unique subscription.

Where multiple Message Brokers exist, subscription information can be shared between brokers to ensure that messages flowing into other brokers can be forwarded to the interested parties. In the case where multiple brokers have subscriptions to a message flow, the updating of subscriptions is also dynamic, with implementation of changes in subscriptions being passed as quickly as a message from one broker to another.

Collectives

When defining brokers within a publish / subscribe network where users may publish information at one broker and other users may subscribe at other brokers it can be more efficient to connect and group brokers together. In MQSeries Integrator, this is termed a Collective. Message brokers are all connected together in a point to point manner, and the

collectives are then interconnected in a tree hierarchy. This allows less individual connections than a fully point to point environment but shorter chains than a tree hierarchy.

The graphic below illustrates a group of systems connected together as collectives.

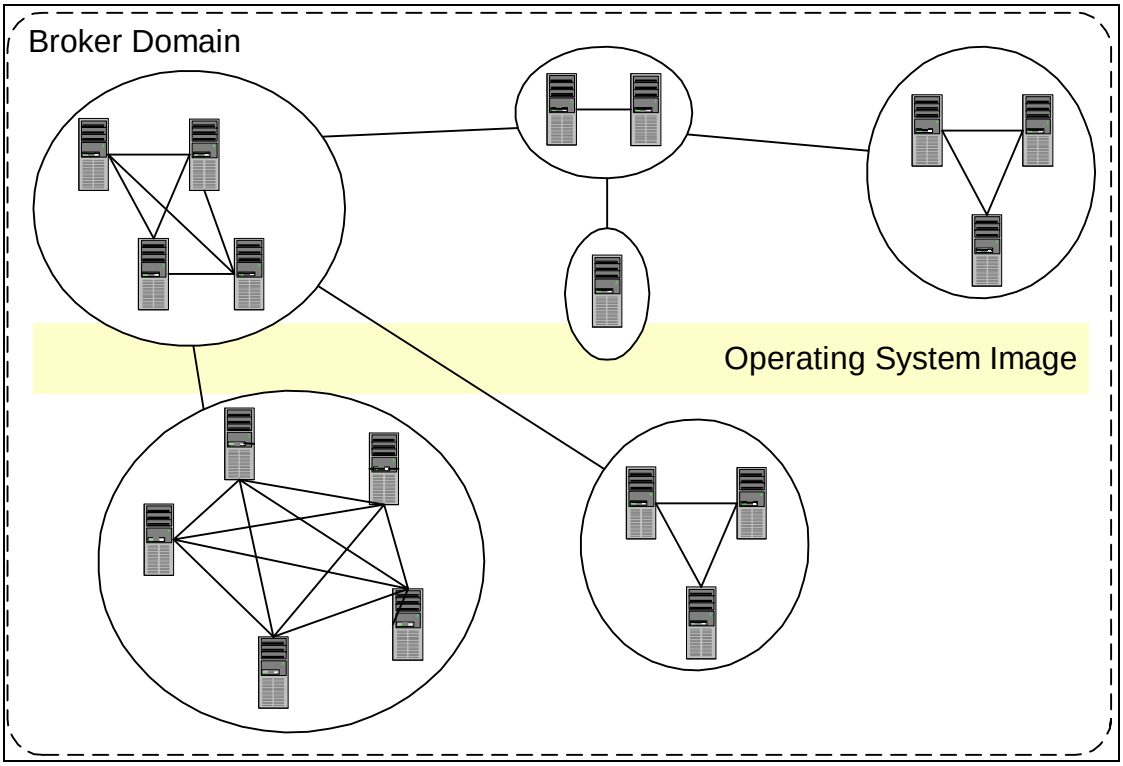

Figure 22 – MQSeries Message Broker Collective

MQSeries Integrator Publish/Subscribe Security

The ability of users to publish information, or subscribe to information depends on the setting of the Access Control Lists (ACLs). The ACLs are set on topics to which the message is published. Publishers must have ACL permission to publish to the required topic. Subscribers must have ACL permission to subscribe to the required topic. Subscribers may request to receive persistent messages, but if denied by the ACLs they will still receive the desired messages, but will not receive them persistently.

Topics are organized into a hierarchical tree structure. The tree structure leads to downstream topics inheriting ACLs from root topics, unless explicitly stated. A subscriber can use wild card topics, and the security policy handles this by applying the policy to the individual leaf topics that are matched by the wild card topic, and ensuring that the ACLs for each particular topic are fulfilled.

The diagram demonstrates the hierarchy of ACLs through the Topics.

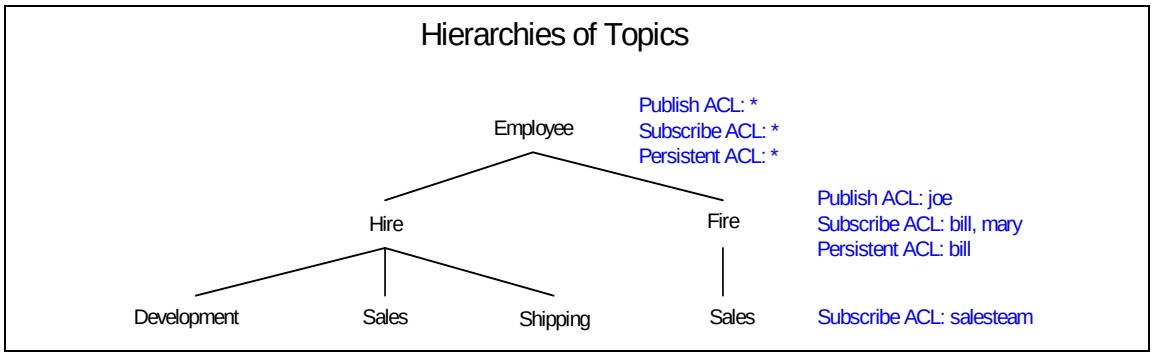

Figure 23 – Hierarchies of Topics

5.5.12. Multi-Broker Domains

Only one Configuration Manager is required in a domain, whatever the number of brokers. There can be more than one **Control Center** to drive the Configuration Manager if required. Multiple message flows can exist in one message broker at any time. Message flows within a broker may be distributed into separate execution groups within the broker, or they may all exist within the same execution group. In addition, multiple brokers may be defined to give the system designer the ability to have many different message flows running on different physical machines at the same time, with each broker having a uniquely identified Queue Manager, shared with no other brokers. Users may also want to be able to have many separate instances of the same message flow to enable greater throughput of similar messages through a specified message flow definition.

5.6. MQSI Control Center

The Control Center, a graphical user interface development tool, is provided with MQSI. This allows the definition of Messages and Message Flows for use in the Broker, as well as the definition and deployment of Brokers and Execution Groups.

In fact all required functions of the Message Broker for all users can be exercised within the Control Center. Multiple concurrent users can use the Control Center. Each user operates within their own workspace. All items to be worked upon by a user are checked out from the Configuration Repository and are then available for update within the user's workspace. This item is then locked to that user for update. When the user wishes to deploy the changes then the item is checked in. When an item is checked in, it is referred to as being shared, as opposed to being in the user's local workspace.

Different classes of users have different views within the Control Center. All operations are available to Superusers. Other classes of users are Message Flow and Message Developers, Message Flow and Message Assignors, Operations and also Security Administrators. The class of the user will give access to different tabs within the GUI for the different permitted operations.

All the information held by the tool is accessed from the configuration repository. This is held in XML format and accessed using the WebDAV protocol. By using the various WebDAV commands the local and shared repositories (WebDAV Resource Servers) are updated and kept in sync. As already mentioned previously the tools that were used in MQSeries Integrator V1 to configure the Rules and Formatter are still shipped with the product, and the use of these tools is still the appropriate way to configure the NEONRules and NEONFormatter nodes. All other functions are controlled using the new GUI.

Format of the Control Center

For a Superuser, with access to all functions of the Control Center, the GUI will look as shown below.

The diagram is a capture of the Control Center while viewing the assignment of message flows to brokers.

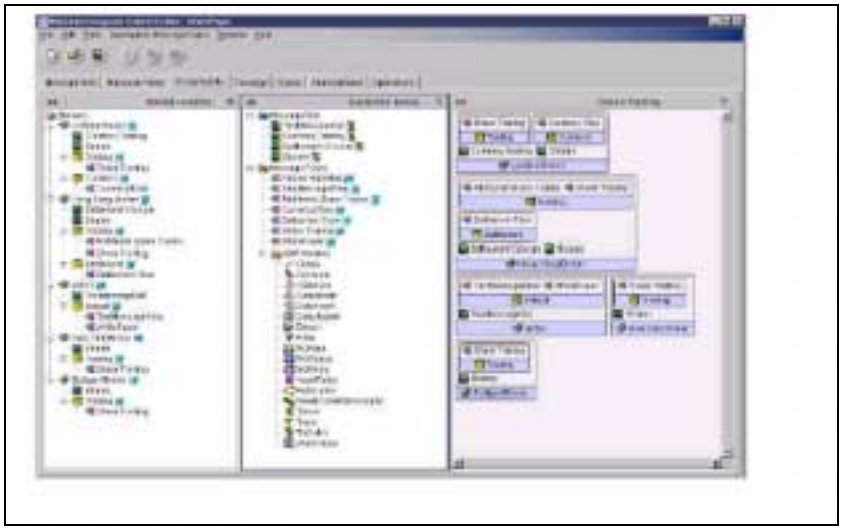

Figure 24 – Control Center – Message Flow Assignment to Brokers

- The *Message Sets Tab* allows users to define for the Message Dictionary the format of Message Sets used within the system. The messages stored within the Message Sets are also defined here, as are the Compound Types used and the Fields and Elements that make up the messages, along with associated Valid Values. The Message Tab integrates into the Message Repository manager, and will allow for full definition of the format and structure of messages. This will include the specification of the Custom Wire Format, and importing and exporting C structures or COBOL copybooks.
- The *Message Flows Tab* displays a split screen, initially showing a palette with IBM Primitives (IBM supplied Nodes) and the workspace to deploy the primitives when defining a Message flow. To define a Message Flow, the system designer specifies a Message Processing Node Type and then drags primitives from the palette onto the right hand part of the screen and connects the nodes by wiring the terminals together. A

defined Message Flow that has been identified as a Message Processing Node Type can then be used as an individual Node in subsequently defined Message Flows.

- The *Assignments Tab* (as displayed in the picture above) allows the System Designer to create and allocate Execution Groups to Brokers. Already created message sets can be assigned to Brokers, and created Message Flows can be assigned to Execution Groups. As has been explained previously in this paper, the Message Flows run within Execution Groups of which there can be a number in a Broker. This tab allows the user to set up each broker, by means of Execution Groups, Message Sets and Message Flows, in the configuration architected by the user. Once satisfied with the selections the configuration can be deployed to the Broker, or Brokers affected.
- The *Topology Tab* is designed to allow a broker's definitions to be registered and be assigned a Queue Manager for deployment to machines where the appropriate installation of MQSeries Integrator has been performed. In addition to creating brokers, this tab also allows for the creation of Collectives. Once a broker has been created within the Control Center, this tab can be used to assign it to a Collective, for it to be a part of a connected Publish / Subscribe Network. Subsequent to assigning a broker to a collective, it can then be connected to a broker in another collective to ensure that the network of collectives is connected together.
- Once assigned, the tab will display all the defined brokers and collectives. In addition, it will display which broker is in which collective, as well as a graphical representation of which broker in each collective is connected to a broker in another collective.
- The *Topics tab* is designed to help the configuration of the Access Control of the Publish / Subscribe Service. There are two alternative views available when using this tab. One view is centered on the Topics and then displays on the right side the access for Groups and Users. Another view is centered on Users and Groups and displays on the right side the Topic Tree accessible by the Users and Groups. By using the functions of this tab the ACLs for any point of the Publish and Subscribe topic tree can be set to allow, deny or inherit the setting for users and groups to publish or subscribe to any topic or sub-topic. The ability to publish, or subscribe to Persistent messages is also set here.
- The *Subscriptions tab* is designed to allow the administrator to display the list of Subscribers and associated subscriptions. This will then allow the administrator to delete subscriptions. This could be useful in the case of a subscriber leaving the company, and wanting to cancel the subscriptions, or in the case of an application failure, preventing any build up of unprocessed messages to an unresponsive application. This tab allows items of information about subscriptions to be tracked. These are Topics, Clients, Brokers, Subscription Points and Registration Date.
- The *Operations tab* provides a basic Systems Management interface for managing the Broker Operations. This tab will allow an authorized user to see whether Brokers, Execution Groups and Message Flows are active or quiesced, and will be able to perform the appropriate operations on these entities. More sophisticated operations than those provided within this tab will be available using plug-in modules available from system management vendors.

6 Application Connectivity (Adapters and Bridges)

6.1. Introduction

The EAI application will be interfacing with several systems. The interfaces between EAI and other systems may require special mechanisms called adapters and bridges.

An adapter or a bridge is a piece of software that moves data between a message on a queue and an application or environment. Adapters handle data inbound-to and outbound-from the application or environment.

For The initial release of the Integrated Technical Architecture, three interfaces were identified. The first interface is the connectivity between EAI and the Component Broker in the Internet space. The second interface is the connectivity of EAI to a CICS legacy system.

This section will discuss the application connectivity requirements for The initial release of the Integrated Technical Architecture:

- MQSeries Application Adapter
- MQSeries-CICS Bridge
- Database Nodes

6.2. MQSeries Application Adapter

MQSeries provides a mechanism for assured delivery of messages, which can be sent even when the target is disconnected. It can be used to distribute work around a large number of disparate systems in an environment where trying to propagate transactional two-phase commit is not practical. The MQSeries application adapter provided by Component Broker is used primarily to provide a semi-transparent integration between Component Broker business applications and applications that based directly on MQSeries. The use of the Component Broker MQSeries application adapter is shown below.

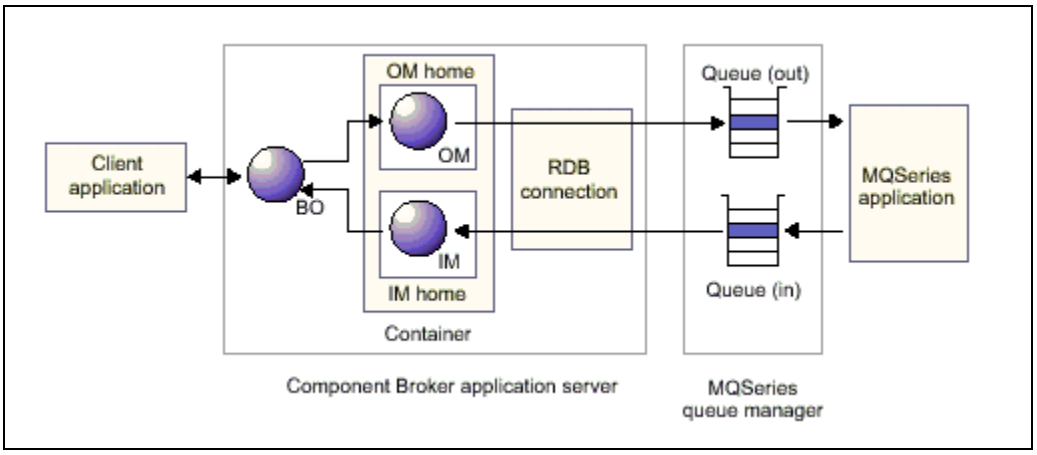

Figure 25 – Component Broker MQSeries Application Adapter

The MQSeries application adapter supports communication between Component Broker servers on Windows NT and Solaris and an MQSeries queue manager on the same local host.

6.2.1. Client application

A client application provides the presentation layer for the application.

It calls the Component Broker business objects to obtain and manipulate data.

6.2.2. Application

A customer-specified implementation of business objects and underlying business logic.

6.2.3. Application server

A Component Broker application server manages applications by instancing the managed objects for applications and providing the services needed to manage access to data and other resources.

An application server can connect to only one MQSeries queue manager during the lifecycle of the server. Therefore, all applications on the server that need to use MQSeries should connect to the same MQSeries queue manager. (All RDB connections for MQSeries provided by applications on the same server should specify the same name for the Open string and Database name properties.) If several applications need to connect to different MQSeries queue managers, you should configure those applications onto different application servers.

6.2.4. Container

Represents and defines the characteristics of a specific queue manager.

A queue manager is treated as just another persistent data store (although with some different characteristics from a relational database) in which data can be stored recoverably and retrieved within the scope of one or more transactions.

6.2.5. Homes

The homes within the container represent the various message formats available for use with a specific queue managed by the queue manager.

OutboundMessage (OM) objects

Managed objects (within an OM home) that represent specific message instances on an outbound queue.

InboundMessage (IM) objects

Managed objects (within an IM home) that represent specific message instances on an inbound queue.

6.2.6. RDB Connection

Used to represent the characteristics of communication with a specific queue manager; for example, the queue manager's name and security policy.

From a client perspective, an MQSeries-backed application looks like any other Component Broker-based client application that uses Component Broker transaction services.

Note: The MQSeries application adapter currently only support transaction container policy; to throw an exception and abandon the call when used outside the scope of a transaction. (Atomic transaction method calls are not supported.)

The managed objects representing messages behave like any other managed object that uses the transaction services. Because such an object represents a message in a queue manager, its life cycle is controlled by the standard messaging data access operations, insert, retrieve, update, and delete (IRUD).

Component Broker, and its MQSeries application adapter framework drive these OM and IM object instances and call the IRUD methods at appropriate times. For example, if a client application is trying to get an inbound message that is not currently instanced in the application server, it creates a new IM object and issues the retrieve method on it to get the message from its queue.

If the client commits the transaction, the message is removed from its queue.

6.3. MQSeries-CICS/ESA Bridge

The MQSeries-CICS/ESA Bridge enables an application, not running in a CICS environment, to run a program or transaction on CICS/ESA and get a response back. This non-CICS application can be run from any environment that has access to an MQSeries network that encompasses MQSeries for MVS/ESA.

A program is a CICS program that can be invoked using the EXEC CICS LINK command. It must conform to the DPL subset of the CICS API that is, it must not use CICS terminal or syncpoint facilities.

A transaction is a CICS transaction designed to run on a 3270 terminal. This transaction can use BMS or TC commands. It can be conversational or part of a pseudo conversation. It is permitted to issue syncpoints.

6.3.1. When to use the CICS Bridge

The CICS Bridge allows an application to run a single CICS program or a 'set' of CICS programs (often referred to as a unit of work). It caters for the application that waits for a response to come back before it runs the next CICS program (synchronous processing) and for the application that requests one or more CICS programs to run, but doesn't wait for a response (asynchronous processing).

The CICS Bridge also allows an application to run a 3270-based CICS transaction, without knowledge of the 3270 data stream. The CICS Bridge uses standard CICS and MQSeries security features and can be configured to authenticate, trust, or ignore the requestor's user ID.

Given this flexibility, there any many instances where the CICS Bridge can be used. For example, when you want:

- To write a new MQSeries application that needs access to logic or data (or both) that reside on your CICS server.
- Your Lotus Notes application to be able to run CICS programs.
- To be able to access your CICS applications from
- Your MQSeries Java client application.
- A web browser using the MQSeries Internet gateway.

6.3.2. How the CICS Bridge works

This section explains how the CICS Bridge works and the options you have when deciding what level of security you want to use.

Points to note in respect of system setup:

- Ensure that the MQSeries-CICS adapter is enabled.
- The CICS Bridge requires that both MQSeries and CICS are running in the same MVS image.
- The MQSeries request queue must be local to the CICS Bridge, however the response queue can be local or remote.
- The CICS bridge tasks must run in the same CICS as the bridge monitor. The user programs can be in the same or a different CICS system.

6.3.3. Running CICS DPL programs

Data necessary to run the program is provided in the MQSeries message. The bridge builds a COMMAREA from this data, and runs the program using EXEC CICS LINK. Figure 2 on page 10 shows the step sequence taken to process a single message to run a CICS DPL program.

The following shows the components and data flow to run a CICS DPL program.

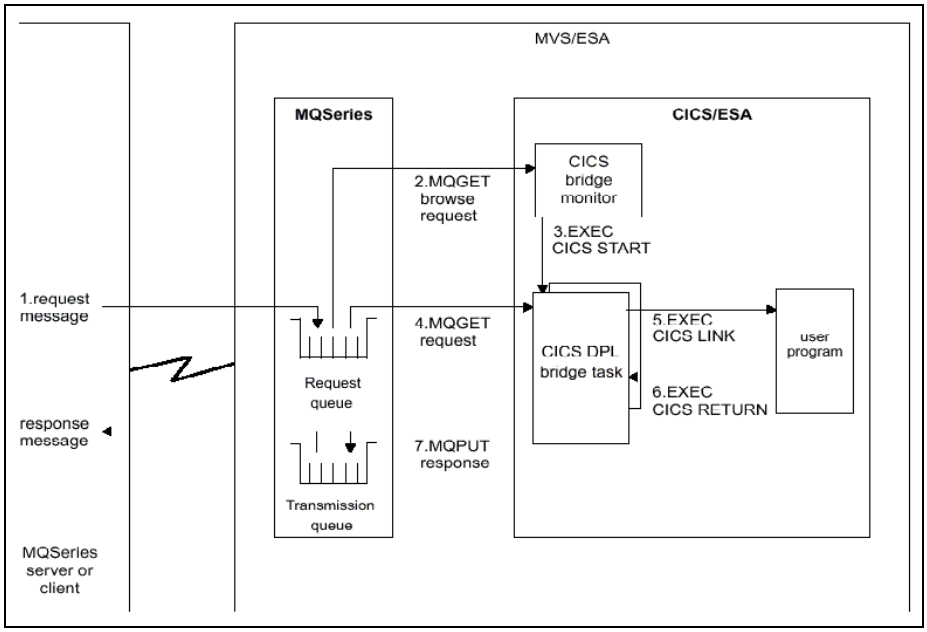

Figure 26 – CICS DPL Transaction

The following takes each step in turn, and explains what takes place:

1. A message, with a request to run a CICS program, is put on the request queue.

2. The CICS Bridge monitor task, which is constantly browsing the queue, recognizes that a 'start unit of work' message is waiting *(CorrelI*d=MQCI_NEW_SESSION).

3. Relevant authentication checks are made, and a CICS DPL Bridge task is started with the appropriate authority (see "Security" on page 12 for more information).

4. The CICS DPL Bridge task removes the message from the request queue.

5. The CICS DPL Bridge task builds a COMMAREA from the data in the message and issues an EXEC CICS LINK for the program requested in the message.

6. The program returns the response in the COMMAREA used by the request.

7. The CICS DPL Bridge task reads the COMMAREA, creates a message, and puts it on the reply-to queue specified in the request message. All response messages (normal and error, requests and replies) are put to the reply-to queue with default context.

8. The CICS DPL bridge task ends.

A unit of work can be just a single user program, or it can be multiple user programs. There is no limit to the number of messages you can send to make up a unit of work.

6.3.4. Running CICS 3270 transactions

Data necessary to run the transaction is provided in the MQSeries message. The CICS transaction runs as if it has a real 3270 terminal, but instead uses one or more MQ messages to communicate between the CICS transaction and the MQSeries application unlike traditional 3270 emulators, the bridge does not work by replacing the VTAM flows with MQSeries messages.

Instead, the message consists of a number of parts called vectors, each of which corresponds to an EXEC CICS request. Therefore the application is talking directly to the CICS transaction, rather than via an emulator, using the actual data used by the transaction (known as application data structures or ADSs).

The following shows the components and data flows to run a CICS 3270 transaction.

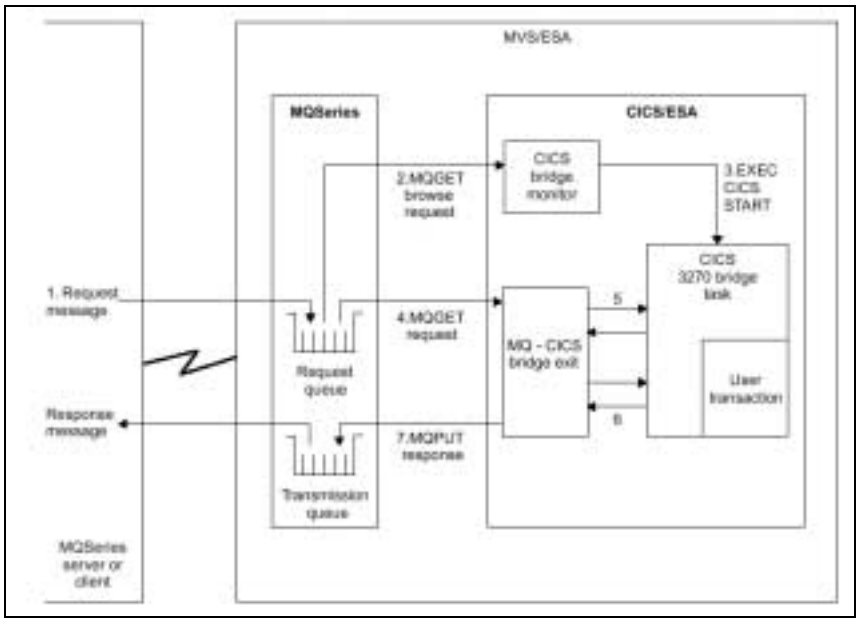

Figure 27 – CICS 3270 Transaction

The following takes each step in turn, and explains what takes place:

1. A message, with a request to run a CICS transaction, is put on the request queue.

2. The CICS Bridge monitor task, which is constantly browsing the queue, recognizes that a 'start unit of work' message is waiting (*CorrelI*d=MQCI_NEW_SESSION).

3. Relevant authentication checks are made, and a CICS 3270 bridge task is started with the appropriate authority (see "Security" on page 12 for more information).

4. The MQ-CICS bridge exit removes the message from the queue and changes task to run a user transaction.

5. Vectors in the message provide data to answer all terminal related input EXEC CICS requests in the transaction.

6. Terminal related output EXEC CICS requests result in output vectors being built.

7. The MQ-CICS bridge exit builds all the output vectors into a single message and puts this on the reply-to queue.

8. The CICS 3270 bridge task ends.

A traditional CICS application usually consists of one or more transactions linked together as a pseudo conversation. In general, the 3270 terminal user entering data onto the screen and pressing an AID key starts each transaction. This model of application can be emulated by an MQSeries application. A message is built for the first transaction, containing information about the transaction, and input vectors. This is put on the queue.

The reply message will consist of the output vectors, the name of the next transaction to be run, and a token that is used to represent the pseudo conversation. The MQSeries application builds a new input message, with the transaction name set to the next transaction and the facility token set to the value returned on the previous message. Vectors for this second transaction are added to the message, and the message put on the queue. This process is continued until the application ends.

An alternative approach to writing CICS applications is the conversational model. In this model, the original message might not contain all the data to run the transaction. If the transaction issues a request that cannot be answered by any of the vectors in the message, a message is put onto the reply-to queue requesting more data. The MQSeries application gets this message and puts a new message back to the queue with a vector to satisfy the request.

6.4. Database Nodes

The EAI application will be interfacing with several external databases, such as Oracle and DB2. This interface is integrated into the EAI system using the Database Nodes supplied with MQSeries Integrator. The Database nodes are specialized nodes that perform a specific function and access a particular database type. The Database nodes, through SQL statements, allow the interaction with the RDBMS via and ODBC interface.

Database operations such as data insert and data update are performed using the Basic node types. SQL statements will be the method for manipulating the data.

6.5. Additional Adapter Requirements

In future releases of the Integrated Technical Architecture, the EAI application will be required to interface with SFA-selected COTS packages, such as Siebel and Oracle Financials. Special, pre-built adapters are available to facilitate such interfaces. Potential implementation options for Siebel and Oracle Financials are shown below.

6.5.1. Siebel Interface

New Era of Networks (NEON) provide an adapter (NEONadapter for Siebel) to interface with Siebel applications.

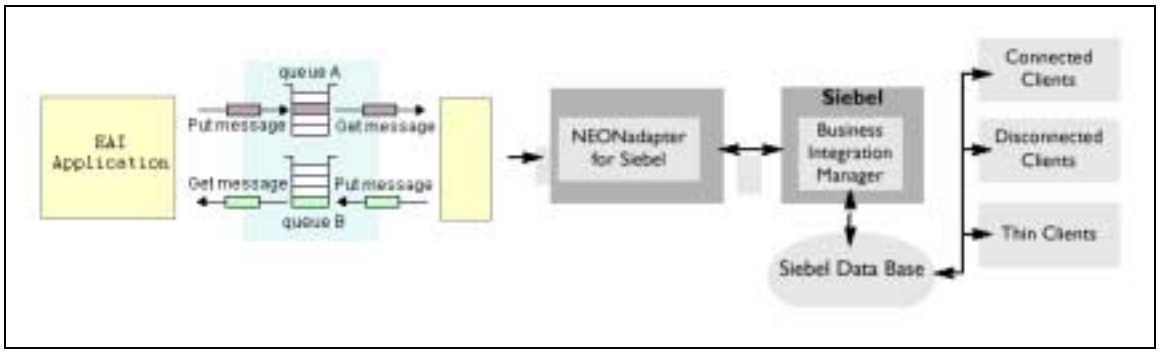

Figure 28 – NEON Interface with Siebel Applications

6.5.2. Oracle Financial Interface

NEON also provides an adapter that may be used for the Oracle Financial interface.

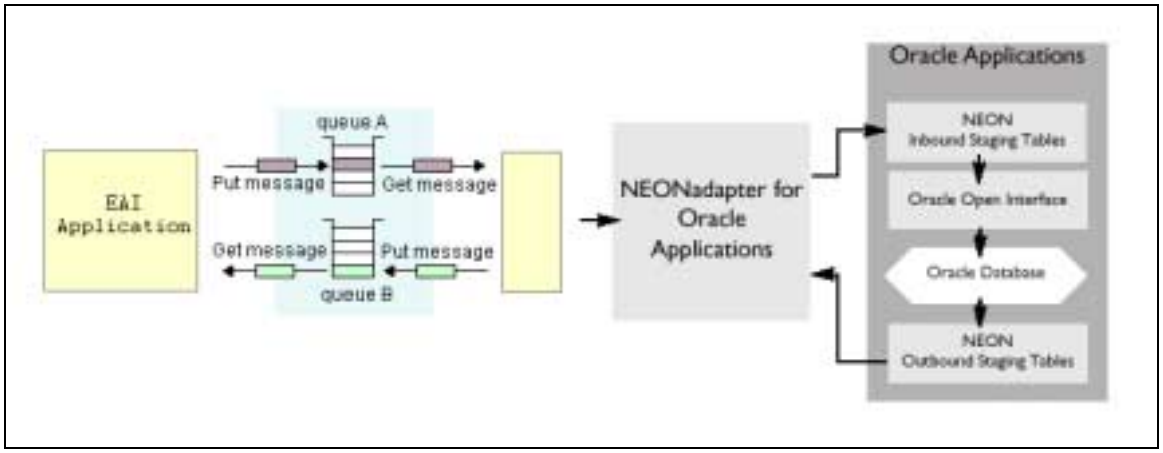

Figure 29 – NEON Interface with Oracle Applications

7 Business Process Management (MQSeries Workflow)

MQSeries Workflow is a workflow management system that will assist SFA in defining and maintaining business processes. MQSeries Workflow provides a process automation system for managing both data and applications.

7.1. MQSeries Workflow Architecture

7.1.1. Scalability

The MQSeries Workflow architecture is highly scalable in numbers of transactions, users and applications. MQSeries Workflow uses IBM DB2 UDB database to store the runtime data for production and MQSeries messaging for its client / server messaging.

MQSeries Workflow is designed to scale vertically and horizontally. By scaling vertically, MQSeries Workflow takes advantage of the processor idle capacity. It also takes advantage of multiple processes on the same system. With this scalability, MQSeries Workflow provides any easier path to migrate from Intel based systems to RISC and mainframe technology.

MQSeries Workflow also scales horizontally by allowing several workflow servers to run using the same runtime database, and by allowing multiple execution servers.

7.1.2. Multi-Tier Architecture

The components of MQSeries Workflow system are designed to be a three-tier structure. The system consists of three logical tiers and two physical tiers.

Tier one: Client components

Tier one represents the client APIs of MQSeries Workflow and the clients that use three APIs. Clients are responsible for executing the program activities that interact with users. The communication with servers is through MQSeries.

Tier two: Server components and Buildtime

Tier two represents the server components and Buildtime of MQSeries Workflow. The server components are responsible for managing the execution of processes at Runtime. The component of the second tier can be distributed across several machines to achieve load balancing. MQSeries is used for the communication between server components as well as between server components and Builtime. The server components can reside on one or more physical machines. The system components that are installed on one physical machine are called a node.

Tier three: Database Server

Tier three represents the Database Server. MQSeries Workflow uses DB2 Universal Database to store the process models and process-relevant data for the System Group. The Runtime database is also involved in the navigation logic between the process steps at Runtime, using SQL calls. This includes status and setup information. The DB2 transport is used for the communication between the Database Server and its clients.

7.2. Transactional Integrity

The MQSeries Workflow production requirements has two opposing requirements for an integrated distributed system:

- High volume and high performance on one side
- Maximum data integrity on the other side.

MQSeries Workflow ensures that no data is lost and the system is in a consistent state, even after system interruptions. Workflow servers' process workflow client requests, status information is stored in the workflow runtime database, and application functions are processed as part of workflow activities. MQSeries Workflow support full transactional semantics on the server side as well as on the application side. It exploits all transactional capabilities of messaging and relational databases on all platforms. For two-phase commit coordination, either MQSeries or the OS/390 Resource Recovery System (RRS) is used. This ensures a maximum of workflow data integrity, and builds upon the transactional strengths and performance of both MQSeries and DB2 UDB.

7.3. MQSeries Workflow Release Schedule

MQSeries Workflow will be installed in The initial release of the Integrated Technical Architecture. However, no business applications will use the workflow components. In future releases, MQSeries Workflow may be used by SFA in the following ways:

- The integration and alignment of the SFA business processes.
- Integration of the applications used by the SFA via the business processes.
- To better accommodate the rapid changes in the SFA requirements
- To enforce and support the execution of SFA business processes such as quality, audibility, and productivity.

8 Queue Design Guidelines

A queue is an MQSeries object owned by a queue manager upon which applications can put or retrieve messages. Applications access a queue by using the Message Queue Interface (MQI). Before a message can be put on a queue, the queue must already exist. Each queue must have a name that is unique to the owning queue manager. Before an application can use a queue, it must open the queue, specifying what it wants to do with it. For example, the application can open a queue to:

- Browse messages only (do not delete them)
- Retrieve messages
- Put messages on the queue
- Inquire about the attributes of the queue
- Set the attributes of the queue

For a complete list of the options related to opening a queue, see the description of the MQOPEN call in the *MQSeries Application Programming Reference* manual.

There are different types of queues. These types include:

- *Local*: a local queue is managed by the queue manager to which the application is connected
- *Remote*: a remote queue is managed by a queue manager other than the one to which the application is connected
- *Alias*: an alias queue points to another queue
- *Model*: a model queue is a template for queue definition
- *Dynamic*: a dynamic queue is a temporary queue defined based on a model queue

In SFA's technical environment, the use of alias queues is discouraged, unless a business need dictates its use (e.g. limiting security access to certain queues). Applications putting messages to remote queues will use the remote queue definition. This allows the application to only specify the remote queue name and not be required to know the remote queue manager name. Model and dynamic queues should be used only when a business need dictates their use.

8.1. Opening and Closing Queues

Before opening a queue using the MQOPEN call, the application must connect to a queue manager. The application can then use the MQOPEN call to open a queue. The application can also then use the MQCLOSE call to close a queue. When an application opens a queue, the application receives an object handle for that queue. This handle is used in subsequent calls to get or put messages. The same queue can be opened more than once; each open call

creates a new object handle. However, most applications will only need to open a given queue once.

Once an application has opened a queue, the application has access to that queue until it closes the queue. The MQOPEN call is costly in terms of time, so once an application has opened a queue and plans to use it in the future, keep the queue open. The exception to this is when an application only needs to put one message. The MQPUT1 call was designed for this case – this call opens a queue, puts the message, and closes the queue, eliminating the need to use the MQOPEN and MQCLOSE calls.

Queues are automatically closed when an application closes its connection to the queue manager. However, it is a good practice to close all queues before disconnecting from the queue manager.

8.1.1. MQOPEN Call

As input to the MQOPEN call, the application must supply:

- A connection handle. Use the connection handle returned by the MQCONN call.
- A description of the object you want to open, using the object descriptor structure (MQOD).
- One or more options that control the action of the call.

The output from MQOPEN is:

- An object handle that represents your access to the queue. Use this as input to any subsequent MQI calls for this queue.
- A modified object-descriptor structure, if the application is creating a dynamic queue.
- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

8.1.2. MQCLOSE Call

As input to the MQCLOSE call, the application must supply:

- A connection handle. Use the same connection handle used to open the queue.
- The handle of the queue you want to close. This comes from the output of the MQOPEN call.

The output from MQCLOSE is:

- A completion code.

- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

8.2. Putting Messages On A Queue

To put messages on a queue, an application must use the MQOO_OUTPUT option when issuing the MQOPEN call. After the queue has been opened using this option, the application can issue an MQPUT call to put a message on the open queue. If the application is only putting one message and will not use the queue again, use the MQPUT1 call.

8.2.1. MQPUT Call

As input to the MQPUT call, the application must supply:

- A connection handle. Use the connection handle that was returned when the application issued the MQCONN call.
- A queue handle. Use the queue handle that was returned when the application issued the MQOPEN call for this queue.
- A description of the message the application is putting on the queue. This is in the form of a message descriptor structure.
- Control information, in the form of a put-message options structure. This options structure needs to be redefined for every MQPUT call.
- The length of the application data contained within the message.
- The application data itself.

The output from the MQPUT call is:

- A reason code.
- A completion code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

8.3. Getting Messages From A Queue

To open a queue so that the messages on that particular queue can be browsed (does not remove the message from the queue), use the MQOPEN call with the MQOO_BROWSE option. To get (and remove) messages from a queue, an application must use the MQOO_INPUT_AS_Q_DEF, MQOO_INPUT_SHARED, or MQOO_INPUT_EXCLUSIVE option when issuing the MQOPEN call. Selection of one of these three options is used to specify if the application opens the queue in exclusive, or shared, mode. See the *MQSeries Application Programming Guide* for more information. After the queue has been opened using

one of these options, the application can issue an MQGET call to get a message from the open queue.

By specifying the MsgId and / or CorrelId fields in the message descriptor structure, the application can search the queue for a particular message. If the application uses MQGET call more than once (for example, to step through the messages in the queue), it must set the MsgId and CorrelId fields of this structure to null after each call. This prevents the call from filling these fields with the identifiers of the message that were retrieved, and therefore getting messages with the same identifiers as the previous message.

If the fields in the message descriptor structure are not specified to search for a particular message, the MQGET call will retrieve the first message in the queue.

8.3.1. MQGET Call

As input to the MQGET call, the application must supply:

- A connection handle. Use the connection handle that was returned when the application issued the MQCONN call.
- A queue handle. Use the queue handle that was returned when the application issued the MQOPEN call for this queue.
- A description of the message the application wants to get from the queue. This is in the form of a message descriptor structure.
- Control information in the form of a get message options structure. This control information describes if the application is browsing or removing messages. The control information also describes if the MQI call waits (and how long it waits) for a message or if the call returns immediately.
- The size of the buffer you have assigned to hold the message.
- The address of the storage location in which the message must be put.

The output from the MQGET call is:

- A reason code
- A completion code
- The message in the buffer area specified, if the call completed successfully
- The options structure, modified to show the name of the queue from which the message was retrieved
- The message descriptor structure, with the contents of the fields modified to describe the message that was retrieved
- The length of the message

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed

9 Queue Manager Design Guidelines

A queue manager supplies applications with MQSeries services. An application must have a connection to a queue manager before it can use the services of that queue manager. An application can make this connection explicitly (using the MQCONN call), or the connection can be made implicitly. For example, CICS for MVS/ESA and CICS/MVS programs do not need to explicitly connect to a queue manager, because the CICS system itself is connected to a queue manager. However, for portability it is recommended that CICS for MVS/ESA and CICS/MVS programs use the MQCONN and MQDISC calls.

9.1. Connecting To and Disconnecting From a Queue Manager

To connect to a queue manager, an application must use the MQCONN call. To disconnect from a queue manager, an application must use the MQDISC call.

9.1.1. MQCONN Call

As input to the MQCONN call, the application you must supply a queue manager name. To connect to the default queue manager, specify a queue manager name consisting entirely of blanks or starting with a null character.

The output from MQCONN is:

- A connection handle. Use this handle in subsequent MQI calls associated with this queue manager.
- A completion code.
- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed. If the reason code indicates that the application is already connected to that queue manager, the connection handle that is returned is the same as the one that was returned when the application first connected. So the application probably should not issue the MQDISC call in this situation because the calling application will expect to remain connected. The MQCONN call fails if the queue manager is in a quiescing state when you issue the call, or if the queue manager is shutting down.

9.1.2. MQDISC Call

As input to the MQDISC call, the application must supply the connection handle that was returned by MQCONN when the application connected to the queue manager.

The output from MQDISC is:

- A completion code.

- A reason code.

Always verify the completion code. If the call is unsuccessful, inspect the reason code for an indication as to why the call failed.

9.2. MQ Series Queue Manager Options

Each physical server MQSeries messaging resides on will have a queue manager located on that server. There are two methods to connect multiple queue managers: Distributed Queuing and Queue Manager Clusters.

- Distributed Queuing

  Distributed Queuing is a method that independent queue managers use to inter-communicate. Typically two queue managers are connected via sender and receiver channels. In addition, a transmission queue has to be defined to send messages from one queue manager to another, and a remote queue definition is required for every queue that resides in the remote queue manager and to which messages are to be sent.

- Queue Manager Clusters

  This facility allows a name to be given to a collection of queue managers, and was introduced in MQSeries Version 5 for AIX, HP-UX, OS/2, Sun Solaris and Windows NT; and MQSeries for OS/390 Version 2.1. It simplifies administration by providing a single system image, and it supports dynamic workload balancing.

  Connecting multiple queue managers using MQSeries clustering facility provides two major functions: (1) simplifies MQSeries object administration and (2) provided workload balancing and failover capabilities. It simplifies the MQSeries administration because it requires fewer object definitions. Transmission queues and Remote Queues are automatically defined and the channel initiator is automatically started when the queue manager starts.

9.3. Queue managers recommendations

A queue manager is in effect a storage container (of queues), and potentially an active daemon managing its content. The first step in the configuration is to start creating the queue managers needed.

9.3.1. Don’t identify any single Queue Manager as the default

Some environments can tolerate an exception, most notably CICS/ESA, where any CICS region is always connected to a single Queue Manager.

Most platforms can have more than one queue manager defined on a system. Don't pick one as the default, as this often results in selecting the wrong queue manager on a particular system.

Even when there is only one queue manager configured, don't define it as the default. Doing so increases the probability for error if another queue manager should be added at a later date.

9.3.2. Pass the connection name as program parameter

This allows a program to run unchanged on any Queue Manager. This provides the capability for multiple concurrent instances; or a queue driven application could be moved to a different queue manager without impacting the application code.

10 MQSeries Naming Standards

This section defines naming standards for MQSeries objects across SFA's enterprise technical architecture. It is recommended that these standards be adhered to when choosing names for new MQSeries objects.

10.1. Common Rules

All MQSeries names should follow MQSeries naming conventions, rather than the standard for object names on each supported platform. Key standards and guidelines:

- Don't use lower case letters

  MQSeries allows both upper and lower case letters in its names. However, MQSeries names are case-sensitive. Using lower and upper case characters for object names is a common source for naming errors.

- Don't use % in names

  This character is valid in all MQSeries names, although it is not commonly used in other names across platforms.

- Choose meaningful names within the constraint of the standard

  Using meaningful names aids the MQSeries Administrator in maintaining the MQSeries environment

  There is no implied structure, or hierarchy, in an object name, such as you might find on many systems' file names. MQSeries just compares the name strings.

  These standards do recommend using hierarchical names under certain conditions. One such example is to use a suffix where there are multiple "versions" of an object.

- Document object names and always include a description

  All objects have a DESCR attribute for this purpose. MQSeries takes no action on the value, but it provides additional information as to the function of the queue.

- Save the definitions

  There are a number of reasons for saving the definitions:

  - In the case of a system failure objects may need to be recreated. To perform this function, the definitions need to be saved separately from the queue manager.

  - They can be used to reset the attributes to a known state. For example if triggering has been turned off, or GET or PUT disabled, it is helpful to be able to restore the objects to their initial state.

  - The definitions can supplement the MQSeries documentation.

10.2. Queue Manager

A queue manager provides the messaging and queuing services to application programs through Message Queue Interface (MQI) program calls. The following guidelines should be followed when naming queue managers:

- **Assign unique names to all queue managers**

  This recommendation can often cause significant problems if queue manager names are not unique. (On MVS, the queue manager name must also be distinct from other subsystem names on the same system.)

  A queue manager can be understood as a "container" for queues and related objects. There is typically one per system, but additional queue managers can be defined.

  Queue Managers with the same name can be configured to exchange messages - by using Queue Manager aliases. This is strongly discouraged. There are some examples where this can lead to ambiguity, and messages can then be sent to the wrong queue manager.

  - If ReplyToQMgr is left blank in the Message Descriptor, MQSeries inserts the actual local Queue Manager name, not its alias.

  - Dead Letter Queue messages identify the real Queue Manager, not any alias.

- Don't copy documentation examples

  Copying the documentation examples provided with the installation files is an easy way to produce queue managers with duplicate names. Plan for the names of queue managers ahead of time.

- Keep the queue manager name short and meaningful

  A recommendation would be to make queue manager names the same as the network host name. However, keep the following points in mind:

  - On MVS, the queue name has to be the same as the host name. The queue manager name corresponds to the MVS subsystem name. Therefore, the queue manager name is restricted to 4 characters.

  - Many queue managers use the first 8 characters when generating unique message identifiers.

  - Channel names, which by convention are derived from queue manager names, are limited to 20 characters.

  - If there is no obvious name, most users would adopt a convention for constructing a queue manager name. Make sure that the convention provides for further expansion, particularly where the restricted names on MVS are concerned.

Some naming examples are illustrated below. A numeric identifier may be appropriate where a processor (or hardware cluster) has multiple queue managers.

Example: CCCDDFNN

CCC = city identifier

DD = company division

F = queue manager function (e.g. Test)

NN = numeric identifier

Example: SSSCCFNN

SSS = stock ticker symbol

CC = city identifier

F = queue manager function

NN = numeric identifier

MVS Example: ADDX

A = geographic area

DD = company division

X = distinguishing identifier

- For a Queue Manager alias, add a suffix to the name

  The main use for this would be to support classes of service. There are fewer constraints on the length of an alias name; it can be more than eight (or four on MVS) characters for example.

  In fact this feature is usually related to defining multiple channels between a pair of queue managers. In this case, use the same suffix for associated channels and queue manager aliases.

10.3. Local Queues

A local queue object defines a local queue belonging to the queue manager to which applications are connected. The following guidelines should be adhered to when naming local queues:

- Local queue names can be up to 48 characters long. They should be short, but long enough to be meaningful.

- Local queue names should not include the name of the queue manager or an indication of the platform used.

- Local queue names should not indicate that the queue is local.

- Local queue names should not include the words local or queue (unless relevant in the context of the application).

- Local queue names should be of the form:

  BusinessUnit.AppName.AppID[.AppID]

  - BusinessUnit is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.

  - AppName is up to 8 characters giving the name of the application using this queue (assigned by the business unit).

  - AppID is a suffix of up to 8 characters giving the application unique identifier (assigned by the application group). If necessary, another identifier (up to 8 characters) may follow the first identifier. These identifiers will be meaningful in the context of the application.

  Examples:

  BU.USCASH.INTRADAY

  BU.CASA.GI0

  BU.USCASH.INTRADAY.NYC

10.4. Remote Queues

A remote queue object identifies a queue belonging to another queue manager. The remote queue is usually given a local definition. The definition specifies the name of the remote queue manager where the queue exists as well as the name of the remote queue itself. The information specified when defining a remote queue object enables the queue manager to find the remote queue manager, so that any messages destined for the remote queue go to the correct queue manager. The following guidelines should be adhered to when naming remote queues:

- Remote queue names can be up to 48 characters long. They should be short, but long enough to be meaningful.

- Remote queue names should not include the name of the queue manager or an indication of the platform used.

- Remote queue names should not indicate that the queue is remote.
- Remote queue names should not include the words remote or queue (unless relevant in the context of the application).

Examples:

A. Destination queue: BU.CASA.GI005

Remote queue name: BU.CASA.GI005

B. Destination queue: BU.USCASH.INTRADAY

Remote queue name: BU.USCASH.INTRADAY.NY

Remote queue name: BU.USCASH.INTRADAY.SS

10.5. Alias Queues

An alias queue object enables applications to access queues by referring to them indirectly in MQI calls. When an alias queue name is used in an MQI call, the name is resolved to the name of a message queue at run time. This enables changes to the queues that applications use without changing the application itself in any way. The following guidelines should be adhered to when naming alias queues:

- Alias queue names can be up to 48 characters long. They should be short, but long enough to be meaningful.
- Alias queue names should not include the name of the queue manager or an indication of the platform used.
- Alias queue names should not indicate that the queue is an alias.
- Alias queue names should not include the words alias or queue (unless relevant in the context of the application).
- Alias queue names should be of the form:

  BusinessUnit.AppName.AppID[.AppID]

  - BusinessUnit is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.
  - AppName is up to 8 characters giving the name of the application using this queue (assigned by the business unit).

- AppID is a suffix of up to 8 characters giving the application unique identifier (assigned by the application group). If necessary, another identifier (up to 8 characters) may follow the first identifier. These identifiers will be meaningful in the context of the application.

Example:

actual queue: BU.USCASH.G3452

alias queue: BU.USCASH.INTRADAY.NYC

10.6. Model and Dynamic Queues

The model queue object defines a set of queue attributes that are used as a template for a dynamic queue. The queue manager creates dynamic queues when an application makes an open queue request specifying a queue that is a model queue. The dynamic queue that is created in this way is a local queue whose name is specified by the application and whose attributes are those of the model queue.

10.6.1. Model Queue Naming Standards

The following guidelines should be adhered to when naming model queues:

- If an application area needs to define a model queue with specific attributes (triggering information or special storage classes), then the model queue name should be of the form:

  BusinessUnit**.**AppName**.MODEL.**AppID**[.**AppID**]**

  - *BusinessUnit* is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.

  - *AppName* is up to 8 characters giving the name of the application using this queue (assigned by the business unit).

  - **MODEL** is a literal indicating this is a model queue.

  - *AppID* is a suffix of up to 8 characters giving the application unique identifier (assigned by the application group). If necessary, another identifier (up to 8 characters) may follow the first identifier. These identifiers will be meaningful in the context of the application.

  Example: BU.USCASH.MODEL.INTRADAY

10.6.2. Dynamic Queue Naming Standards

Dynamic queue names should be of the form:

BusinessUnit**.**AppName**.DYNQ.**AppID**[.**AppID**]**

- *BusinessUnit* is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.

- *AppName* is up to 8 characters giving the name of the application using this queue (assigned by the business unit).

- **DYNQ** is a literal indicating this is a dynamic queue.

- *AppID* is a suffix of up to 8 characters giving the application unique identifier (assigned by the application group). If necessary, another identifier (up to 8 characters) may follow the first identifier. These identifiers will be meaningful in the context of the application.

Example: BU.USCASH.DYNQ.INTRADAY.REPLY

10.7. Transmission Queues

A transmission queue temporarily stores messages that are destined for a remote queue manager. Transmission queues must be defined for each remote queue manager that a local queue manager will send messages to. It is possible to associate several transmission queues with different characteristics with a remote queue manager. This allows different classes of transmission service. The following guidelines should be adhered to when naming transmission queues:

- Transmission queue names will include the name of the adjacent (i.e. directly connected) queue manager. The transmission queue name will be the name of the destination queue manager only in the case where the destination queue manager is directly connected with the sending queue manager. Otherwise, the transmission queue name will be the name of some other queue manager that will play the middle party in a multi-hop message transfer to the destination queue manager.

- If there is only one channel to the queue manager, use the exact name of the adjacent queue manager.

- If there will be multiple channels to the queue manager, use the adjacent queue manager name followed by a dot and some class of service.

- If the exact queue manager name is not used, appropriate queue manager alias definitions need to be provided to allow MQSeries to perform queue manager name resolution.

- Transmission queue names should be of the form:

  AdjacentQueueManagerName**[.**ClassOfService**]**

Examples:

SFANYC_QM

SFANYC_QM.BATCH

10.8. Dead Letter Queues

A dead-letter queue (also known as an undelivered-message queue) receives messages that cannot be routed to their correct destinations. This occurs when, for example:

- The destination queue is full
- The message cannot be put on the destination queue
- The sender is not authorized to use the destination queue
- The destination queue does not exist

The following guidelines should be adhered to when naming dead-letter queues:

- Dead-letter queues should be of the form: CITI.DLQ
  - **CITI** is a literal standing for Department of Education . This is included to remain consistent with the business unit identifier of naming standards for other queue types. If multiple business units share the same queue manager, there can be only one dead-letter queue. Therefore, the literal CITI is used as a common business unit name qualifier.
  - **DLQ** is a literal standing for the dead letter queue.

  Example: CITI.DLQ

10.9. Initiation Queues

An initiation queue receives trigger messages, which indicate that a trigger event has occurred. A trigger event is caused by a message that satisfies the specified conditions being put onto a queue. Messages are read from the initiation queue by a trigger monitor application that then starts the appropriate application to process the message. If triggers are active, at least one initiation queue must be defined for each queue manager. The following guidelines should be adhered to when naming initiation queues:

- Initiation queue names should be of the form:

  BusinessUnit**.INITQ.[**Identifier**]**

- *BusinessUnit* is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.

- **INITQ** is a literal standing for the initiation queue.

- *Identifier* is up to 8 characters giving an optional unique identifier for this initiation queue. This identifier will be a CICS region name for CICS regions, the literal BATCH for MVS batch work, or a meaningful string for other platforms.

Example: CITI.INITQ.BATCH

10.10. Processes

A process definition object defines an application to an MQSeries queue manager. Typically in MQSeries, an application puts or gets messages from one or more queues and processes them. A process definition object is used for defining applications to be started by a trigger monitor. The definition includes the application ID, the application type, and application specific data. The following guidelines should be adhered to when naming processes:

- Process names should not include the name of the queue manager or an indication of the platform used.

- Process names should not indicate that the object is a process.

- All process names should be of the form:

  BusinessUnit**.**AppName**.**AppID**[.**AppID**].PROCESS**

  - *BusinessUnit* is up to four characters indicating the name of the business unit using this queue. This high-level qualifier will be useful when applications from multiple business units share the same machine / queue manager.

  - *AppName* is up to 8 characters giving the name of the application using this queue (assigned by the business unit).

  - *AppID* is a suffix of up to 8 characters giving the application unique identifier (assigned by the application group). If necessary, another identifier (up to 8 characters) may follow the first identifier. These identifiers will be meaningful in the context of the application.

  - **PROCESS** is literal indicating this MQSeries object is a process definition.

  Example: BU.CASA.GI005.RECEIVE.PROCESS

10.11. Channels

A channel provides a communication path. There are two types of channel, message channels and MQI channels. A message channel provides a communication path between two queue managers on the same, or different, platforms. The message channel is used for the transmission of messages from one queue manager to another, and shields the application programs from the complexities of the underlying networking protocols. A message channel can transmit messages in only one direction. If two-way communication is required between two queue managers, two message channels are required.

An MQI channel connects an MQSeries client to a queue manager on a server machine. It is for the transfer of MQI calls and responses only and is bi-directional. A channel definition exists for each end of the link. The following guidelines should be adhered to when naming channels:

- Channel names can be up to 20 characters long.
- Channel names should be of the form:

  SendingQM**.**ReceivingQM**[.**ClassOfService**]**

  - *SendingQM* is the name of the sending queue manager (without the _QM).
  - *ReceivingQM* is the name of the receiving queue manager (without the _QM).
  - *ClassOfService* is optional and is used to distinguish between different classes of service between the same two queue managers. This qualifier is limited to the number of characters remaining after the sending and receiving queue manager names have been combined to form the channel name.

  Based on the above channel naming standard, channel names can always be interpreted as *FromQueueManager.ToQueueManager* without ambiguity.

  Examples:

  SFANYC.CITISS

  SFASS.CITINYC.BATCH

10.12. MQSeries Integrator

MQSeries has been enhanced with enterprise application integration (EAI) functionality. MQSeries Integrator supplies rules-driven routing and data transformation , which simplifies the task of integrating diverse applications across the enterprise. MQSeries Publish and Subscribe supports routing of topic-based messages to dynamic subscribers based upon the content of the message. These two facilities are compatible, and can be used to construct complex messages and routing based on business logic.

The same general principles can also be used when naming queues associated with these functions:

- Subscriber queues are in fact application input queues, so application naming standards apply to these queues.
- MQSeries Integrator has input queues, which can be given a hierarchical name – just as if the EAI tool was an application, and provides the first part of the queue name.

11 Application Interface Programming Options

There is a wide range of options for communicating with MQSeries programs including new interfaces for message content as well message delivery. Programs written using any of these message delivery styles can communicate with each other, and with programs written in any of the other MQ delivery styles.

11.1. Message Delivery

11.1.1. MQI

The Message Queue Interface (MQI) is the common API across all platforms. The calls made by the applications running on each platform are common. This allows application programmers to focus on the business logic of the application, rather than the interface differences of each platform. This makes it much easier to write and maintain applications, as well as facilitate migration of applications from one platform to another as required by changing business needs.

The following figure represents the MQI.

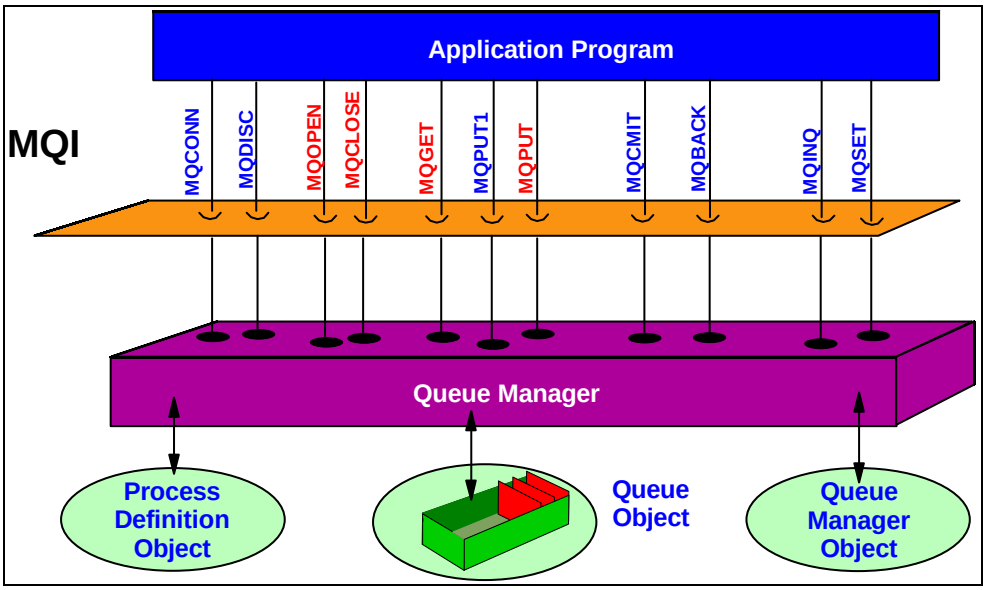

Figure 30 – Message Queue Interface

11.1.2. JMS

Java Message Service (JMS) is supported by an MQSeries implementation of this Java standard API for Enterprise Messaging Services. Using JMS, applications can communicate

with other MQSeries JMS applications, with applications written to the MQI, or to the Application Message Interface (AMI).

11.1.3. AMI

The MQSeries AMI can be used to build client applications, and the AMI will automatically build any required headers as specified using the AMI, including the new RFH2 headers. The AMI is designed to simplify the task of the application programmer, while enabling the more advanced functions and message broker facilities to be used.

AMI is a high level API that moves many functions normally performed by messaging applications into the middleware layer, where a set of policies defined by the enterprise is applied on the application's behalf. Policies hold details of how messages are to be handled, for example, priority, confirmation of delivery, timed expiry. IBM provides a suite of common policies, and an open policy handler framework that allows additional policies to be created by third-party software vendors or the enterprise.

11.2. Message Content

11.2.1. XML

eXtensible Markup Language (XML) is an industry-wide standard for self-defining messages. It enables diverse systems and databases to understand each other's data (for example, to identify fields) by indicating both the content and the role of the data.

XML is supported in MQSeries Integrator Version 2 and MQSeries Workflow Version 3.2; XML will be supported within MQSeries Messaging via the Common Messaging Interface.

11.2.2. CMI

Common Message Interface (CMI) is a logical message construction API, used in conjunction with a message delivery API (for example, MQI, AMI, or MQSeries Support for JMS). It dynamically constructs and parses messages, interrogating and modifying them as appropriate. It supports XML messages, and can use a message dictionary to validate message formats or substitute default field values, for example.

12 EAI Common Error Handling Guidelines

Whenever possible, the queue manager returns any errors as soon as an MQI call is made. The three most common errors that the queue manager can report immediately are described in this section.

12.1. Failure of an MQI Call

An example of an MQI call failure is being unable to put a message to a queue because the queue is full. The completion code and return code of the MQI call specify the nature of the failure. Applications should inspect these codes for every MQI call and be able to handle all possible return codes.

12.2. System Interruption

The queue manager is an example of a system component needed by the application and when the queue manager is interrupted, the application encounters an error. Applications must ensure no data is lost due to this sort of interruption. To ensure no data loss, applications will get and put messages under syncpoint. This syncpoint activity can be controlled by the queue manager or by some external resource coordinator (e.g. CICS, Encina, etc.).

12.3. Unable to Process Messages

Messages containing data that cannot be processed successfully are known as poisoned messages. When applications operate under syncpoint, if the application cannot successfully process a message, the MQGET call is backed out. The queue manager maintains a count (in the BackoutCount field of the message descriptor) of the number of times this happens. Messages whose backout counts increase over time are being repeatedly rejected by the application – the application should be designed to handle such situations.

12.4. Responding to Errors

Applications should respond in a similar manner to errors returned by MQI calls. One possible way to implement this common error handling methodology is to provide error-handling routines for the application developer. Use of these common error-handling routines ensures that all application programmers handle MQSeries errors in the same way and do not have to write their own error handling routines.

13 EAI Operations Environment Considerations

This chapter describes some operational considerations for the EAI architecture, including ways in which to monitor MQSeries components.

13.1. Stopping queue managers

If a queue manager in the message broker needs to be stopped, avoid stopping it immediately in the first instance, or the result can be that it is subsequently slower to restart.

A better approach is to quiesce the queue manager first. Well-behaved channels and connected programs are expected to detect this condition; they may continue in order to preserve work already done, but are expected to disconnect within a reasonable time. (Five minutes is usually considered an acceptable maximum.) The queue manager stops when all programs have disconnected.

In particular, avoid canceling distributed queue managers. A checkpoint is written when a queue manager is ended by command. Canceling a queue manager would not take that checkpoint, and so it would need longer to restart.

13.2. Dead Letter Queue

If MQSeries can detect an error synchronously, it is reported directly to the application. If a message can not be delivered after that it is a candidate for the Dead Letter Queue. This preserves a message that can not be processed immediately, without stopping valid messages in the meantime.

The facility is available on all platforms except MQSeries for Windows V2.

MQSeries for AS/400 documentation refers to it as the “undelivered-message” queue, but is otherwise the same.

Although normally described as a channel function, there are other MQSeries components that write to the Dead Letter Queue, including Trigger Monitors and the IMS Bridge.

Include a Dead Letter Queue on all queue managers.

On all queue managers, use a local queue called SYSTEM.DEAD.LETTER.QUEUE. This is created automatically by some MQSeries platforms. On those platforms that do not, create a queue with this same name; it will cause less confusion to use a common name everywhere.

It is still necessary to configure the queue manager, by identifying this queue in its DEADQ attribute. If a Dead Letter Queue is required, and is not available, a channel will fail.

Some users have avoided defining a Dead Letter Queue in order to detect errors sooner, but that is not recommended. The problem with this approach is that one rogue message is sufficient to stop all messages across a channel.

Consider ways to avoid unnecessary DLQ messages.

Some platforms allow an automatic retry if a message can not be delivered immediately. It is specified by parameters on a receiving channel, and the conditions can be changed through a Retry Exit. The channel is paused while such retry is in progress. Thus, transient errors can be tried again to avoid messages being written to the Dead Letter Queue unnecessarily.

Outside the scope of this document, note that applications can also request undelivered messages are discarded, using the MQRO_DISCARD option; often used in combination with Exception With Full Data, to return the message to its sender.

Process the undelivered messages.

Messages that are put on the Dead Letter Queue take the form of the original message data, preceded by a **dead letter header** - defined by the MQDLH structure. The header includes the intended destination queue, and queue manager, for the message, and the Reason it could not be delivered.

Listing the contents can be sufficient for a test system. A production environment such as a message broker must have a process, triggered or scheduled at intervals, to dispose of the messages appropriately. Some platforms supply a Dead Letter Queue Handler (rules driven); otherwise you would need a program for this purpose.

Construct rules based on queue names, message type, feedback code, etc. It can be appropriate in some cases to retry or discard certain messages.

Where no such action is appropriate, transfer the undelivered message to an application-related queue for action there. A reasonable default action on a message broker would in fact be to transfer the undelivered message to the **Dead Letter Queue** on the appropriate application queue manager.

13.3. Making channels run faster

Where applicable, define channels with MCATYPE(THREAD).

On Windows NT and OS/2, message channel agents can run as a Thread or a Process. Version 5.1 has extended this capability to UNIX as well. The channels must be started through the MQSeries Channel Initiator or Listener for the specified choice to take effect. Using threads would result in lower system overhead.

Consider defining multi-threaded agents.

CSD #3 for MQSeries Version 5 enhances the support for multi-threaded agents by allowing the number of agent processes (AMQLAA0) attached by the queue manager to be set by statements in the **TuningParameters** stanza of the QM.INI file.

The advantage of multithreaded agents is that they significantly reduce the number of agent processes. Instead of creating another agent process for every application, an additional thread is added to an existing agent process.

Details can be found in MQSeries SupportPac MP02.

As a last resort, consider trusted Listeners and Channels.

This option needs to be treated with some care, and after other methods have been exhausted. If it is used it applies to all channels in a queue manager; and would be used instead of the multi-threaded agents described above. It can also lead to severe problems if the conditions set out below are not met.

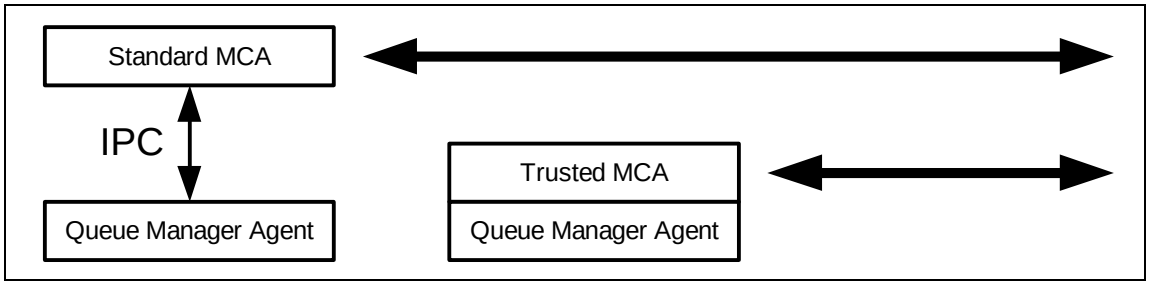

Figure 31 – Trusted Bindings

Trusted bindings were introduced as an option in MQSeries Version 5, and were also made available through a CSD on some version 2 products.

Channels designated as trusted run in the same process as their queue manager agent, and so avoid the IPC overhead that was really intended as a protection from potentially errant applications.

This reduces the number of processes required, and removes unnecessary overhead of inter-process communication. There is no loss of integrity as a result because these components, of MQSeries, can be trusted not to fail and compromise the queue manager and its applications as a result.

Some results suggest somewhere between a 2 and 3 times improvement over non-persistent with NPMSPEED(NORMAL); and not much difference against NPMSPEED(FAST). The advantage of being trusted is in faster PUTs and GETs, particularly of persistent messages, so this is where you might see the most benefit.

This is not the default. It needs to be specified in the channel stanza of qm.ini to request it, so it applies to all channels in a queue manager.

```
MQIBindType=FASTPATH
```

Channels with trusted binding must not be stopped with MODE(FORCE).

Ending a Trusted Channel in a non-controlled manner is likely to result in the queue manager being left in an unstable state and needing to be recycled. In this context, stop mode(force) is regarded as an unexpected end and should not be used.

If any trusted application (channels included) is ended without closing its open objects and disconnecting from the queue manager in the prescribed manner, MQSeries will not be able to free the associated resources.

CSD 4 changes the way a receiving MCA works from a blocking to a non-blocking socket call (this is TCP/IP specific, of course), so it should always be possible to close a channel successfully without using mode (force).

13.4. Monitoring queue managers on MVS

MQSeries for MVS/ESA provides ways to monitor a queue manager in production, and has parameters for tuning its overall performance as a result.

13.4.1. Page set usage

Measure use of space within a page set.

Use of space in a page set can be tested with the following command:

```
DISPLAY USAGE
```

If queues need to be moved to a different page set based on this information (or to match the use of buffer pools better), follow the instructions in “Managing page sets”, in the System Management Guide.

13.4.2. SMF 115

Monitor actual use by collecting SMF 115 records

SMF type 115 records collected during operation provide a base for determining if buffer pools are an appropriate size, in addition to the checkpoint frequency mentioned earlier. Where Service Level Report (SLR) is in use at installation, extend its usage to cover MQSeries.

SupportPac MP15 can help format the results here.

Tune the buffer pool sizes based on results.

Check the SMF type 115 records for the following points particularly.

In all buffer pools, the size needs to be large enough to avoid synchronous page access. QPSTSOS particularly needs to remain zero, and also QPSTDMC.

Where messages are expected to be taken from queues soon after being put there, the size of the buffer pool needs to be large enough to minimize disk access. Aim for the following:

- Low ratio of QPSTRIO to QPSTGETP
- Low ratio of QPSTSTL to (QPSTGETP+QPSTGETN)

Where messages are processed some time after being put, increasing the size of the buffer pool will have little effect. Since there would be more initialization time, there could be a negative result. It need to be large enough to prevent serious disk contention, but otherwise kept small so that there is a steady offload.

13.4.3. Checkpoints

Have an appropriate checkpoint frequency.

The LOGLOAD default (10,000) provided in CSQ6SYSP is normally much too low for heavy use.

Buffer pool pages older than two checkpoints are written to disk. Fifteen minutes or more would be typical for a high use system; consistently more than ten per hour is too many. Taking checkpoints too infrequently can result in longer restart time after a failure though.

QJSTLLCP in the log manager statistics of SMF 115 records indicates the number of checkpoints that have been taken.

13.4.4. CICS adapter

If CICS adapter problem, check TCBs

The MQSeries CICS adapter has 8 TCBs for handling MQ requests, and this number is usually sufficient. It can be a bottleneck though on high volume systems where there are long MQGETs – needing and extended search or disk access.

The CKQC transaction can display the status. If the number of busy TCBs is always at the maximum, there is a SupportPac, which can provide further information

13.4.5. Other factors

Is Message Retry causing excessive delay?

Many queue managers support a message retry function in receiving MCA channels. The intent is to wait and retry if there is a transient delivery problem, rather than to fail messages immediately. Other messages on the channel are held up while this waiting takes place.

The default considers Queue Full and Put Inhibited as transient; the action is to wait 10 seconds, and to repeat up to 1000 times if the error remains. (The default action can be changed by a channel retry exit, and the exit is invoked for any type of delivery error.)

If these errors really are transient there is value in pacing a channel this way to prevent messages arriving on the DLQ unnecessarily. Frequent or lasting errors would clearly have an adverse effect on the channel's performance – but fix the real errors rather than remove message retry.

Would compression exit help?

When large messages are sent over a slow network there can be value in adding compression in Send and Receive channel exits. This needs to be a trade off against the overhead of invoking the exit.

No compression exit is supplied with the product, but there is a sample available on AIX, OS/2, Windows and Windows NT; it can be found in the MQSeries

SupportPac MO02: MQSeries message compression support.

13.5. MQSeries for Sun Solaris Startup Procedures

This section describes how an MQSeries queue manager should be started in a Sun Solaris environment provided that the underlying communications protocol is running and that the communication links are active. Some of the events that must be started are:

- Start the Sun Solaris queue manager (strmqm qmgrname)
- Start the Sun Solaris channel initiator (runmqchi –m qmgrname –q initq)
- Start the Sun Solaris command sever (strmqcsv qmgrname)

Start-up automation on Sun Solaris platform

The commands above should be added to the Sun Solaris "startup" file for the queue manager, channel initiator, and the command server to automatically start when Sun Solaris is booted.

MQSeries for Sun Solaris Shutdown Procedures

This section describes the order in which the MQSeries Queue Manager and its components must stop in order to assure graceful shutdown and faster restart.

Before the shutdown of a queue manager is issued ensure that no runmqsc process is running. This will prevent the queue manager from shutting down.

To shutdown a queue manager a controlled shutdown (endmqm –c) should be issued. MQSeries will signal all "well behaved" user applications that the queue manager is terminating. This procedure will ensure that all outstanding units of work are completed in the manner intended by the application prior to ending the queue manager, with no units of work in-doubt. Any active channels and the channel initiator should terminate normally.

NOTE: A "well behaved" application is one that uses the "FAIL_IF_QUIESCING" option on MQOPEN, MQPUT1, MQGET, and MQPUT calls. When a controlled shutdown (endmqm –c) is issued these applications will get a return code of QMGR_QUIESCING. This way the application has the opportunity to commit the work done to that point or rollback the current unit of work to a consistent state.

If the shutdown is very slow or if the queue manager does not appear to be stopping then an immediate shutdown (endmqm –i) can be issued. If the immediate shutdown still doesn't stop the queue manager then a pre-emptive shutdown (endmqm –p) should be used.

If the queue manager has not shutdown after this sequence is complete then use the procedures described in the "Stopping a Queue Manager Manually" in the MQSeries System Administration manual.

NOTE: KILLING THE PROCESSES IN THE WRONG ORDER CAN CORRUPT THE QUEUE MANAGER, REQUIRING RE-INSTALLATION OR RECOVERY FROM A BACKUP WITH LOSS OF DATA.

If the queue manager shutdown command used was different from a controlled shutdown (endmqm –c), it is recommended to record a media image of the objects by issuing the MQSeries command "rcdmqimg". This writes in to the linear log enough information to completely recreate an existing object.

MQSeries for Sun Solaris Backup Procedures

For integrity of the data one should use "Linear Logging" to be able to use media recovery to restore damaged objects. Linear logs create new files as each one is filled and processed. MQSeries log's disk usage will grow indefinitely. Therefore, a procedure must be in place to archive and delete the old log files. Periodically, the queue manager writes a pair of messages in the … / mqm / qmgrs / qmname / errors file to indicate which of the log files is still required for media recovery and to restart the queue manager.

Message AMQ7467 gives the name of the oldest log file needed to restart the queue manager. This log file and all newer log files must be available during queue manager restart.

Message AMQ7468 gives the name of the oldest log file needed to do media recovery.

Any log files older than the ones specified in the messages can be archived to tape and deleted from the … / mqm / log / active directory.

It is strongly recommended to put the logs (… / mqm / logs) onto a different physical drive from the one used for the queues (… / mqm / qmgrs). Log files should be kept on different devices from the queue manager files for performance and system integrity. They should be placed in multiple disk drives in mirrored arrangements.

A backup procedure should be scheduled for at least once a week. Backup the queue manager file directories (…mqm / qmgrs) and log file directories (…mqm / logs), including all respective subdirectories. In addition backup the log control file, the MQSeries configuration file, and the queue manager configuration file. On Sun Solaris, use the "tar" command to backup the data and make sure that the ownership of the files is preserved. The queue manager MUST be down during the backup process.

NOTE: The queue manager file directories and the log file directories should be backed up at the same time to ensure they have the same ages and that they are in the same state. In addition, it is strongly recommended to backup the directories that may be found empty. They will be required when restoring the backups in the future.

MQSeries for Sun Solaris Recovery Procedures

This section describes what procedures to use depending on the type of failure that occurs in the system.

- Communication failure
- A damaged MQSeries object
- A damaged log

Communications failure: If a communications failure occurs messages being transmitted to a remote queue will remain on the transmission queue until they can be successfully transmitted. To recover from a communications failure it might be sufficient to just stop and restart the channel that used the link that failed.

To ensure that a channel ends normally after a communication failure it is recommended to configure the channel with the following attributes:

Disconnect Interval (DISCINT) – It is a channel attribute. If no messages arrive on the transmission queue during the specified time interval, the channel closes down. Next time a message arrives on the transmission queue the channel initiator will automatically start the

channel. A very low value (a few seconds) may cause excessive overhead in constantly starting up the channel.

Heartbeat Interval (HBINT) — It is a channel attribute. When there are no messages on the transmission queue, the sending MCA will send a heartbeat flow to the receiving MCA, thus giving the receiving MCA an opportunity to quiesce the channel without waiting for the disconnect interval to expire. The heartbeat interval value should be significantly less than the value specified in the DISCINT. HBINT is valid on the SENDER, RECEIVER, SERVER, REQUESTER, and CLUSTER channels.

TCP / IP KeepAlive – It is defined in the MQSeries QM.INI configuration file in the … / mqm / qmgrs / QMGRname. A stanza must be set to enable TCP / IP KeepAlive. SO_KEEPALIVE is an option on the TCP / IP socket. If you specify this option, TCP / IP periodically checks that the other end of the connection is still available, and if it is not, the channel is terminated.

A damaged MQSeries Object: If the queue manager encounters a damaged MQSeries object during startup the queue manager automatically tries to recreate it from its media images, if linear logging is used. If any of the defined queues can not be recovered the queue manager will NOT start. Manually delete the file containing the damaged object and restart the queue manager. Media recovery of the damaged object is automatic.

If a single object is reported as damaged during normal operation, recreate the object from its media image by issuing the "rcrmqobj" command.

If the queue manager object has been reported as damaged during normal operation, the queue manager performs a preemptive shutdown. Manually delete the file containing the damaged queue manager and restart the queue manager. Media recovery of the damaged queue manager object is automatic.

A damaged log: It is strongly advised to use disk mirroring for the MQSeries log files in order to minimize the risk of not being able to recover persistent messages.

Note: If the queue manager and log backups need to be restored, restore them at the same time to ensure that their ages are the same and that they have a valid state.

MQSeries for Sun Solaris System Management

To simplify the management of an MQSeries environment, standards and naming conventions should be defined. In addition, the MQSeries environment and its objects must be protected against unauthorized access through some form of security management. To ensure that the health of the MQSeries environment is adequate monitoring and administration procedures and tools must be evaluated.

- Naming Convention
- MQI Security

- Monitoring and Administering MQSeries

Naming Convention:

An MQSeries object naming standard should be developed. See the MQSeries Object Naming Standard document for more information.

MQI Security:

MQSeries for Sun Solaris supplies an Object Authority Manager (OAM) that exploits the security feature of the underlying Sun Solaris operating system. In particular the OAM uses Sun Solaris user and group Ids. The OAM manages users' authorizations to manipulate MQSeries objects such as queues, process, and channels.

The MQSeries API issues resource security checks during MQCONN, MQOPEN, MQPUT1, and MQCLOSE. When a program issues an MQOPEN or an MQPUT1 call using an alias queue name, MQSeries uses the alias queue name to check authorization. It does not use the resolved queue name (target queue name).

Monitoring and Administering MQSeries:

Each MQSeries queue manager provides a separate MQSeries environment consisting of MQSeries objects, trigger monitors, and MQSeries configuration files(mqs.ini and qm.ini). MQSeries for Sun Solaris uses a number of error logs to capture error messages concerning the operation of MQSeries itself, any running queue manager, and error data coming from channels in use. When an error occurs and the queue manager name is known the error message is logged under that queue manager's directory (… / mqm / qmgrs / qmgrname / errors). When an error occurs but the queue manager name is not known the error is logged under the @SYSTEM subdirectory (… / mqm / qmgrs / @SYSTEM / errors).

Different types of errors, warnings, and other significant occurrences related to a queue manager cause events to be entered in a specific event queue, which can be used to initiate an automated response.

- SYSTEM.ADMIN.QMGR.EVENT – Event messages put in this queue are related to the resources within a queue manager. For instance, an event message is generated when an application attempts to put a message to a queue that does not exist.
- SYSTEM.ADMIN.PERFM.EVENT – Event messages put in this queue are notifications that a threshold condition has been reached by an object. For instance, a queue depth limit has been reached.
- SYSTEM.ADMIN.CHANNEL.EVENT – Event messages put in this queue are conditions detected by a channel during its operation. For instance, when a channel is stopped.

A centralized MQSeries monitoring tool may use the SYSTEM.ADMIN.EVENT queues to monitor the MQSeries network. A monitoring tool will report availability and performance

of queue managers, channels, and queues. It will allow operations of channels and queues, browse error logs, and browse dead letter queue from a single point of control.

MQSeries for Sun Solaris Performance and Tuning

There are a few MQSeries customization issues that can affect the performance of an MQSeries application.

- The allocation of log files
- The allocation of buffers

The allocation of log files: Log files should be kept on different devices from the queue manager files for performance and system integrity. They should be placed in multiple disk drives in mirrored arrangements. Logs and queue manager files should be located in different file systems and different physical devices.

Linear logs create new files as each one is filled and processed. MQSeries log's disk usage will grow indefinitely. Therefore a procedure must be in place to archive and delete the old log files. Periodically, the queue manager writes a pair of messages in the …/mqm/qmgrs/qmgrname/errors file to indicate which of the log files is still required for media recovery and to restart the queue manager.

The allocation of log buffers: There are two parameters used to tune the log buffers (LogFilePages and LogBufferPages). These parameters can be specified in the queue manager configuration file (qm.ini). The greater the number of persistent messages in the system the more critical these parameters become.

MQSeries Performance and Tuning in a Distributed Environment

Batch size: Message throughput is very dependent on batch size. A batch size is an attribute of the sender, receiver, and cluster channels. A batch size is determined by the message arrival rate in the transmit queue. A low batch size may cause the transmission queue to build up and a high batch size will make little difference. Unless there are throughput or communications link issues that require frequent commits, set a high batch size (the default is a good value to be used) and let it dynamically adapt.

Fast Messages: The fast message option improves the performance of non-persistent messages by removing the overhead of commit processing on the channel. Fast message is an option feature on each channel and it causes non-persistent messages to be lost in case of a channel failure.

MQSeries User Exits

This section describes how MQSeries allows addition of functions through the use of exits. There are a few user exits available with MQSeries. Channel Exits are exits specified in the channel definition.

- *Security Exit*: This exit is called during channel setup. The purpose of this exit is to allow additional code that can check security credentials of the partner Message Channel Agent (MCA). This exit can be used for non-security purposes as well. The support pack provided by IBM uses this exit to change some of the control information in the message header.
- *Message Exit*: This exit is called just after a message is read from the transmission queue and just before a message is put onto its destination queue. It receives the entire message and message descriptor (MQMD). It allows the message and message descriptor to be changed. It is typically used to encrypt and decrypt a message data.
- *Send Exit*: This exit is called just before a buffer of data is transmitted over the network. It receives only the message data. It is typically used to compress data.
- *Receive Exit*: This exit is called just after a buffer of data is received from the network. It is typically used to decompress data.
- *Data Conversion Exit*: This exit is called during the process of an MQGET call using option MQGMO_CONVERT. The format field in the message descriptor (MQMD) is used to define the name of the loadable object containing the exit. It is used to convert byte ordering in integers and character sets and encoding in character strings used in different machines.

Detailed information on how to use the channel exits and the data conversion exit is defined in the IBM MQSeries Application Programming Guide.

13.6. MQSeries for MVS/ESA Startup Procedures

This section describes how an MQSeries queue manager and related components should be started in an OS/390 environment provided that the underlying communications protocol is running and that the communication links are active. The three major components required to be started are:

- Start the MVS Queue Manager Subsystem (START QMGR PARM(parmname))
- Start the MVS Channel Initiator Address Space (START CHINIT(parmname))
- Start the MVS Listener (START LISTENER TRPTYPE(TCP) PORT(port#))

The command server does not need to be started. It is started automatically when the queue manager starts.

It is important to wait until TCP/IP, the Queue Manager, and the Channel Initiator have all started before starting the Listener. The “START CHINIT” command should be coded in the CSQINP2 DD concatenation file of the queue manager started task procedure. This way when the queue manager starts it will automatically start the channel initiator.

A user-supplied dataset containing the “START LISTENER” command can be part of the CSQINPX DD concatenation files of the channel initiator started task. When the channel

initiator address space (CHINIT) starts it will execute the "START LISTENER" command defined in the CSQINPX file.

The MQSeries Triggering facility should be used to automatically start a Sender Channel when a message arrives in the transmission queue.

The MQSeries-CICS attachment facility can be automatically started, via PLTPI with parameters specified in the INITPARM of the CICS start-up Proc or in the CICS system initialization table (SIT). Specify the name of the queue manager, initiation queue, and a trace number to identify the adapter in CICS trace entries in the parameter list.

MQSeries for MVS/ESA Stop Procedures

This section describes the order in which the MQSeries Queue Manager and its components must stop in order to assure graceful shutdown and faster restart.

Stopping MQSeries

- Before stopping MQSeries make sure all MQSeries console messages have received their replies.
- STOP LISTENER TRPTYPE(TCP): This will stop the TCP / IP listener process.
- STOP QMGR MODE(QUIESCE): This will terminate the MQSeries queue manager only when ALL connection threads have ended. MQSeries will signal all "well behaved" user applications that the queue manager is terminating. This procedure will ensure that all outstanding units of work are completed in the manner intended by the application prior to ending the queue manager with no units of work in-doubt. Any active channels and the channel initiator address space should terminate normally. If MQSeries is archiving a log the "STOP QMGR" command will not take effect until the archiving has finished.

  NOTE: A "well behaved" application is one that uses the 'FAIL_IF_QUIESCING" option on MQOPEN, MQPUT1, MQGET, and MQPUT calls. When the STOP QMGR MODE (QUIESCE) is issued these applications will get a return code of QMGR_QUIESCING. The application has the opportunity to commit the work done to that point or rollback the current unit of work to a consistent state.
- DISPLAY THREAD(*): This command displays active threads. It shows batch jobname(s) and / or CICS region(s). If there are no active threads and the MQSeries does not terminate then issue "STOP QMGR MODE(FORCE)".

  NOTE: DO NOT CANCEL MQSeries address space unless STOP QMGR MODE (FORCE) does not terminate the queue manager.

MQSeries for MVS/ESA Backup Procedures

This section describes backup procedures that can minimize any future recovery problems. The MQSeries datasets are:

- Bootstrap dataset (BSDS) – BSDS datasets contain an inventory of all active and archived log datasets.
- Pagesets – MQSeries provides one hundred pageset datasets. Pageset zero is used to store all object definitions required by a queue manager. All other pagesets ranging from 01 through 99 are used to store messages. Storage class (a parameter of a local queue definition) maps one or more queues to a page set.
- Active Logs – Active Logs contain information needed to recover persistent messages and MQSeries objects.
- Archive Logs - Archive Logs are copies of Active Logs. When an Active Log fills up MQSeries copies its contents to a DASD or tape dataset called Archive Log.

Specify dual Active Logging, dual BSDS datasets, and switch archiving on (OFFLOAD=YES) in the MQSeries macro CSQ6LOGP of the MQSeries subsystem initialization parameters (CSQZPARM). These may not be necessary in a development environment, but it is required in a QA and Production environments to avoid loss of data. Each copy of the Active Logs, each copy of the BSDS datasets, and the pageset datasets should reside in separate physical volumes and if possible, in mirrored arrangements.

Each time a new Archive Log is created a copy of the BSDS is put into the Archive Log. If an Archive Log dataset is deleted the information about the Archive Log must be removed from the BSDS dataset using the CSQJU003 utility.

Consider having two backup copies of each back-up cycle. Make backups of the MQSeries BSDS, pagesets, and the corresponding log datasets at least once a week to obtain a weekly point of recovery and for disaster recovery purposes. Ensure that the RBA number in page 0 of each page set, called the recovery log sequence number (LSN) is backed-up. This number is the starting RBA in the log from which MQSeries can recover the page set. Provided that all logs are available from this point forward all messages can be recovered to the point of failure.

MQSeries for MVS/ESA issues two messages to assist in managing the logs.

- CSQI024I -- This message gives the restart RBA (relative byte address) for the subsystem, but does not include any offline page sets in the calculation of this restart point.
- CSQI025I -- This message gives the restart RBA (relative byte address) for the subsystem, including any offline page sets.

All log records must be kept as far back as the lowest RBA identified in messages CSQI024I and CSQI025I.

To obtain a full backup of the MQSeries datasets the queue manager must be shutdown during the backup process. If the queue manager is running during the backup process updates may be held in buffers, which means the backup datasets are NOT in a consistent state. That is called a “fuzzy” backup. Please see the MQSeries for MVS/ESA System Management Guide for information on the steps required to create a full back up.

MQSeries for MVS/ESA Recovery Procedures

This section describes what procedures to use depending on the type of failure that occurs in the system.

- Queue Manager ended abnormally
- Pageset dataset is full
- Pageset dataset failure
- BSDS recovery

Queue Manager ended abnormally:

An abnormal termination can leave data in an inconsistent state leaving units of recovery in doubt. MQSeries resolves the data inconsistencies during restart.

Pageset data set is full

A pageset is a LINEAR VSAM dataset that is formatted to be used by MQSeries. Up to 123 secondary extents can be defined for a pageset provided that enough disk space is available. The pageset utilization can be displayed with MQSeries command DISPLAY USAGE PSID(nn). The pageset zero should be used for systems entries ONLY. An MQSeries queue manager will not start if the pageset zero becomes full or is unavailable.

When a pageset is full an application program may receive a reason code of MQRC_PAGE_SET_FULL from an MQI call. At this point the pageset can either be expanded or the messages load balanced between multiple pagesets.

A queue manager must be down in order to expand a pageset. A pageset is expanded by creating a new pageset, formatting it, and copying all the messages from the old pageset to the new one. CSQUTIL provides FORMAT and COPYPAGE functions that can be used to expand a pageset.

Pageset dataset failure

Provided that all needed recovery logs are available MQSeries can recover a pageset during restart. If the logs are not available the point of recovery backups can be used to recover a pageset. Please see “Recovering pagesets” in the IBM MQSeries for MQS/ESA System Management Guide.

BSDS recovery procedures

The BSDS dataset MUST be a dual dataset residing on different volumes. Dual BSDS dataset mode is specified in the CSQ6LOGP macro which is part of the systems initialization (CSQZPARM). If one BSDS gets damaged MQSeries changes to a single BSDS mode and MQSeries continues running without a problem. The damaged BSDS copy needs to be recovered before restart. To recover a damaged BSDS use the VSAM AMS utilities to delete

the old BSDS and to create a new BSDS VSAM cluster. In addition the MQSeries RECOVER command must be used to copy the active BSDS into the newly created BSDS and to reinstate the dual mode.

If both BSDS datasets get damaged the last Archive Log can be used to recover the BSDS datasets. Please read the step by step procedure listed in the IBM MQSeries System Management Guide section "BSDS Recovery".

MQSeries for MVS/ESA System Management

Security: MQSeries for MVS / ESA uses the MVS system authorization facility (SAF) to interface to an External Security Manager (ESM), such as RACF. The ESM manages users' authorizations to manipulate MQSeries objects such as queue manager, queues, process, and channels.

13.7. MQSeries Integrator System Management

13.7.1. Installation

There are a number of components to be installed before the system can be configured and used. This list of components will vary depending on which platform is being installed. A full list of components and hardware and software requirements will be available in the product documentation and the announcement letter.

If required, the installation tool can also be used to uninstall the product, removing both the product code base and any entries in the system registry database.

13.7.2. Configuration and Set-up

Once the product has been successfully installed, it needs to be configured before use. The tasks that product configuration will need to perform are as follows:

Definition of message flows

This requires the designing of the components along with building new components by wiring together existing components.

Definition of message sets/formats

This will define the logical definition of a message format and the assignation of it to a message set.

Definition of brokers and broker topology (including execution groups)

This will set the association between the message flow and an execution group, and also the association between message sets and defined brokers.

Definition of publish/subscribe topology within a broker

This will cover all the connectivity issues between brokers and collectives to propagate publish and subscribe messages across a broker domain.

Definition of Access Control to topics and policies Security for users and groups is handled by the underlying operating system security.

Publish / Subscribe topics are defined to MQSeries Integrator and principals (groups and / or users) are associated with them. ACLs for publish / subscribe and other information are then associated with principals.

Deployment of the defined configuration

Once a configuration has been defined, it should then be deployed to the specified broker to load the runtime configuration. The configuration manager process will be notified if this deployment fails.

13.7.3. Interfaces for Definition and Deployment

All configuration changes are routed through the configuration manager. This is required to enable the configuration manager to provide information to recover / restart any set-up that needs to be recreated, even if the changes have only been applied during runtime. Communication between user interfaces and the configuration manager, and between the configuration manager and the broker components, uses MQSeries Messaging.

The messages for the various interfaces are designed to be best suited to the needs of those using the interfaces. For example, the monitoring and reporting interfaces are defined as a set of XML messages that are published by the MQSeries Integrator broker, along with an associated set of system meta-topics. These can be subscribed to by applications needing to monitor the state of the broker. As appropriate to the information it will either be retained or be available on a request / reply model.

13.7.4. Deployment of Changes

As already stated, for all brokers in a domain, the configuration data is held by the configuration manager, which is unique in a domain. This holds two stores of data, one of which is the shared working version and one of which is the deployed, or runtime, version.

The deployed version is populated with data from the shared working version by *deploying* the data. By working with the Control Center, the shared working data can be deployed. Authorized local users of the machine on which it is installed can only use the Control Center.

Individual objects cannot be selectively deployed, but entire sets of objects instead must be deployed. These sets could be any or all of the following:

- All relevant message flows, execution groups, message sets
- All topics and associated ACLs
- All Pub / Sub topology information

However, the deployment can be selected to deploy just the changes rather than redeploy the entire configuration, thus improving the performance of the deployment. In order to deploy the data to the brokers, the format of the XML as used by the tools must be *compiled* into the XML format understood by the broker. Once this is complete the data is transmitted to the broker using a message to the Administration Queue for that broker, where the Administrative Agent in the broker handles it.

The agent deploys the data to the affected components. These update the active runtime cache and activate the changes. Messages on the success of the update will be processed through the Administrative Agent and then the Configuration manager, where the report can be further processed.

As an example, when changing a Message Dictionary that is already available in a Resource Manager, a new version of the dictionary is deployed, using WebDAV for versioning. Thus an older version number can identify the version that might be held in a RTD than that held by the Resource Manager. New dictionaries and new dictionary versions are inserted into broker administration messages and published to all relevant brokers using broker administration messages. The dictionary can then be passed on to the RTD in order to replace the existing deployed version.

14 EAI Performance Tuning

As more and more application functions are connected together within the EAI technical environment, performance tuning will become critical in ensuring the overall success of the ITA.

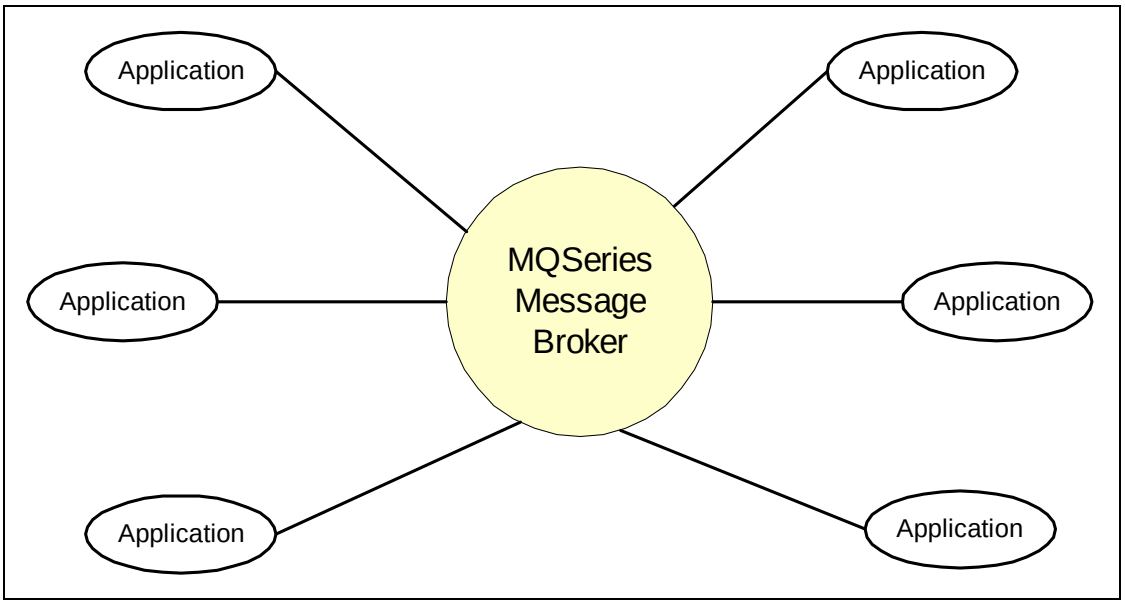

Figure 32 – MQSeries Message Broker

The need to design for performance is evident from the fact that all enterprise messages are required to flow through the message broker in the ITA. The need for the message broker to "perform well" can be generalized to include such topics as reliability, or to allow changes without disruption to applications.

This section does not discuss the performance design of programs that run using the message broker, except where it can be affected by configuration options. For this purpose the assumed characteristics of programs in the broker are as follows:

They are designed to process a stream of messages efficiently; they do not connect to the queue manager or open queues repeatedly for example.

- They are well behaved, in the MQSeries sense. If the operator shuts down the queue manager, the programs can continue to a point of consistency, but would end cleanly within a reasonable time (5 minutes is usually regarded as an acceptable limit).

- They do not perform application function; nor any long-running process on messages.

14.1. Requirements

Some design requirements on the message broker performance can be gathered from assumptions about the applications, and the messages that flow between them. For example:

- The size of the messages, and the rate they flow through the broker – and whether it is a sustained rate, or some other arrival pattern.
- Whether there is an application requirement for the broker to transfer messages in a defined time, or if messages could be accumulated and delivered as a batch later.
- Whether messages are persistent or not.

14.2. Dynamic Workload Distribution

Dynamic Workload Distribution (DWD) is a new capability in MQSeries for OS/390 V2.1, and in MQSeries Version 5.1. DWD allows clusters of Queue Managers, across multiple systems, that share definitions of public queues and channels, yet look like a single, local system.

The message broker and connected application nodes might be regarded as a messaging cluster, even though the application queue managers do not communicate directly. The benefits are enormous:

- Heavy workloads can be balanced across multiple Queues and/or systems.
- The system is inherently and transparently fault-tolerant.
- Administration of complex MQSeries networks is simplified.
- Conflicts between departments sharing MQSeries resources are eliminated

Dynamic Workload Balancing also offers strong failure-tolerance in its normal mode of operation, making it very cost effective in terms of hardware and administration. By having more than one instance of the same queues in a cluster, each performing an equivalent function, a failure leaves the others to continue unaffected. This can also be combined with fail-over capability from other high-availability technologies.

14.3. Capacity planning information

The most important and current source of information for capacity planning is the MQSeries SupportPac library. It contains detailed performance reports from the product group; the reports usually arrive shortly after the product is generally available, and contain planning information not found in the documentation.

http://www.software.ibm.com/ts/mqseries/txppacs/txpm3.html#perfor

14.4. Designing Queue Managers for Performance

The first place to look for performance benefits is the queue manager located on the message broker. Some parameters can be changed to improve performance, but some are fixed when the queue manager is created. As a result, review this section before creating a production queue manager.

14.4.1. Logging

Logging of persistent messages is commonly the key activity which would limit local queuing performance, and particularly if the messages are large.

Most MQSeries implementations, and certainly those likely to be found in a message broker, include logging for recovery. The following queue managers are the exceptions that do not perform logging.

If logging is a constraint, extra queue managers can help.

If a queue manager is expected to have many large persistent messages, its constraint may be writing to the log disks. If there is processing speed in hand, extra capacity may be achieved through multiple queue managers on the system. This does require each queue manager to have its own dedicated log drive to achieve this end; and any message channel communication needed between these queue managers would have to be included in the estimate.

Address the logging configuration ahead of time.

Logging is often found to be a limiting factor in overall MQ performance. If its configuration is not correct when it comes to tuning an existing system, the best approach is to define a new queue manager following these recommendations.

Decide the type of logging to support the level of recovery required.

Circular logging is simpler to manage. It provides recovery from system and network failures, but not from disk failures where the queues reside.

Additional integrity may be achieved using mirrored disks to replicate queue or log storage. On MVS/ESA, MQSeries can be configured to write duplicate logs.

- Ensure log drives are separate from queues. On any system, configure MQSeries so that logging is performed on a drive that is separate from queues. This is necessary anyway if the logs need to recover from disk failures. In addition, MQSeries will be faster by avoiding the contention.
- On MVS/ESA keep log data sets separate from each other. Dual logs need to be on separate drives, from each other as well as from the page sets containing the queue data.

- Each log should have at least two log data sets, and ideally more in a production system. These need to be on separate drives too, in order to avoid disk contention during log archive.
- On MVS/ESA do not switch archiving off in a production system.
- Where there is a choice, put logs on the fastest drive available. On most systems this means aiming for a dedicated disk, or at least one that is lightly used. On MVS/ESA, the log disks can be configured with caching and fast write.

Note that MQSeries on distributed platforms needs to force log writes to disk immediately, and verifies that the file system will support this feature. Some operations occasionally require an immediate write to queue files. As a result, MQSeries generally will not allow a network drive to be used for logs or queues.

14.5. UNIX kernel parameters

Below are some current recommendations for the UNIX kernal parameters for MQSeries Version 5.

- If the existing value is already higher than what is shown (for other software on the system), leave it unchanged. Use the system default or existing value for other parameters.
- These recommended values might need to be higher if there are multiple queue managers on a system, or if there are many MQSeries channels.
- Increase **semmni** by 50 for MQSeries V5. There has been a significant reduction from MQSeries V2, where the original documented value should be used. Review the actual usage by applications since this can be expensive.
- Similarly increase **semmns** by 200 for MQSeries V5.
- Otherwise, the recommendation would be to set the following:

  semmnu = 2048

  semume = 256

  shmmax = 0X30000000

  shmmni = 512

  shmseg = 1024

14.6. MVS Queue Managers

This section contains some further points to verify an MQSeries queue manager is set up right. Unlike logging, these are specific to a platform; but they are factors that can be tuned in a production queue manager, and in response to observed performance data.

14.6.1. Page sets and storage class

Never map queues to page set zero.

In MVS/ESA MQSeries stores messages and queue definitions in linear VSAM data sets known as page sets. In operation, MQSeries operates on this data in memory buffer pages, and manages the disk access to page sets when needed.

The page sets are numbered 00 – 99, and MQSeries stores object definitions and attributes page set zero. There will be a performance impact due to the contention if queue data is assigned there as well. In addition, there is a danger of serious problems if page set zero ever becomes full.

Verify that no storage class identifies page set zero.

Local Queues are defined with reference to a storage class; each storage class is assigned to a page set. Hence there is a level of indirection. One possibility is to define application specific storage classes to enable page set reselection. Use the following command to check that no storage class maps to page set zero.

DISPLAY STGCLASS(*) PSID(00)

The aim would be to have to no storage class listed as meeting this selection. Even if a storage class is listed there may be no corresponding queue in page set zero; in which case change the assigned page set, or delete the storage class if not needed.

If queues do need to be moved from page set zero, follow the instructions in “Managing page sets”, in the System Management Guide.

14.6.2. Buffer pools

Plan buffer pool selection to enable tuning.

The buffer pages are arranged in up to four pools; the size of each is configured with the DEFINE BUFFPOOL command in the CSQINP1 initialization input data set, and takes effect when the queue manager is started.

The size of each buffer pool can be used for tuning performance. For the moment, here is a suggested differentiation of the four pools in order to offer most scope.

Reserve for use with page set zero. The documentation recommends not putting messages in the same buffer pool as the object attributes.

Short life messages, which are expected to be transferred out of the message broker immediately after arrival. A large buffer pool here would tend to keep messages resident in memory, and avoid access to disk.

Long life messages. Where messages are stored for later delivery there is little to be gained trying to keep messages in memory. A smaller buffer pool to ensure a steady offload would be quicker to recover if needed.

Default, or messages where performance is less critical – and keep them separate from the previous ones with known requirements.

Page sets are assigned to buffer pools when processing the CSQINP1 data set. Plan to enable the most appropriate buffer pool selection in restarting the queue manager.

Assign each queue to its optimum buffer pool.

The section above described the need to configure a queue manager so that Buffer Pools can be optimized for different queuing characteristics. The corresponding recommendation for applications or a message broker is to understand the characteristics of messages in its queues, and to assign them to appropriate storage. Some users employ an application specific storage class to enable separate tuning later if needed.

14.6.3. Indexed queues

Define a queue as indexed where appropriate.

If any queue is primarily retrieved by MsgId or CorrelId, MQGET times can be significantly improved by defining the queue with the corresponding Index. Note that it can take time to build (at restart) an index for deep queue, one containing many messages, since the entire queue has to be read.

14.7. Channels

This chapter discusses the design of the MQSeries channels, which connect the message broker to nodes. The performance of the channels is frequently the main factor in overall performance of the message broker. In addition to these design choices, some external factors will govern MQSeries channel capacity, including for example: speed and utilization of the network; speed of MQSeries log devices at both ends; CPU speed at both ends. There may be further operational factors such as whether messages from the transmission queue will need to be read from disk.

14.7.1. Classes of service

In general, messages with different characteristics would need to transfer through the message broker, and it is sometimes useful to separate them into “classes of service”. For example, sending very large messages for later processing could impact the response time of shorter messages. There may be other reasons too, such as security.

The way to do this with MQSeries is to have multiple channels to separate the classes of service.

14.7.2. Number of channels

Memory requirement often limits the total number of channels. Apart from the capacity needed to support the message transfer rate, the usual limiting factor for the number of active channels is the amount of memory each needs. The SupportPac library can give more detail, but the following is a rough guide for MVS channels.

Each MVS channel requires some reserved storage "below the line". This was around 8K bytes in early releases; about 1200 bytes in version 1.2; still less in the current version 2.1.

The virtual memory needed for each MVS channel is about 150KB plus the maximum message size. The availability of paging space is usually more limiting in current releases than below line storage.

Configure one pair of channels between two queue managers.

Apart from the need to separate classes of messages to different channels in this way, one reasonable question is whether multiple channels between two queue managers would perform better than one.

The result however is that defining extra channels increases the storage requirement, and makes less effective use of the batch capability.

14.8. Network tuning

MQSeries can operate over different types of network, the choice being transparent to the applications. Some MQSeries configurations include multiple types of network. There are some performance differences between the network types, but in many cases the transport type will have already been chosen for reasons other than MQSeries.

The network that is used to carry the messages may also be regarded as a channel component, and as such has some further potential for tuning.

14.8.1. SNA

RUSIZE is typically not enough.

MQSeries is often configured over existing networks, where the RUSIZE typically has a value of 256, which is sufficient for terminal connection. This is not enough to run MQSeries effectively, so use an MQ specific LOGMODE with a higher value. The SupportPac performance reports use an RUSIZE of 1920 throughout, and it should be higher still for larger messages.

The parameter may appear under a different name on other platforms, but this recommendation applies just the same.

The pacing parameter is often too low.

A value of 20 is suggested. Better still, specify adaptive pacing where possible, and let SNA determine the best parameters. Ensure pacing is compatible at both ends.

Specify DELAY = 0 in the VTAM PU definition.

It has been reported that setting this value can significantly increase throughput and reduce response time.

14.8.2. TCP/IP

Increase the MTU Size.

The SupportPac report recommends increasing the MTU size, specified on the GATEWAY statement, to 65572.

15 Further Information

- MQSeries home page, http://www.software.ibm.com/ts/mqseries/
- MQSeries Planning Guide, GC33-1349
- MQSeries Queue Manager Clusters, SC34-5349
- MQSeries Intercommunication, SC33-1872
- MQSeries for MVS/ESA V1.2 System Management Guide, SC33-0806
- MQSeries for OS/390 V2.1 System Management Guide, SC34-5374
- MQSeries System Administration, SC33-1873
- MQSeries SupportPacs

These are supplementary materials freely available from the web to enhance MQSeries, including reports and samples. The text identifies a link to the catalog of all that relate specifically to performance. The following, including some listed other than in the performance category, are particularly relevant to the scope of this document.

- MD01: MQSeries – Standards and conventions
- MO02: MQSeries message compression support
- MP02: MQSeries Version 5 – Multithreaded Agents
- MP15: MQSeries for MVS/ESA – Printing statistics and accounting records

16 Acronyms

Table 1 – List of Acronyms

Acronym	Description
AMI	Application Message Interface
API	Application Programming Interface
CICS	Customer Information Control System
CMI	Common Message Interface
CORBA	Common Object Request Broker Architecture
COTS	Commercial-Off-the-Shelf
DOE	Department of Education
DWD	Dynamic Workload Distribution
EAI	Enterprise Application Integration
EJB	Enterprise Java Bean
GB	Gigabyte
IBM	International Business Machines
IRUD	Insert, Retrieve, Update, and Delete
ITA	Integrated Technical Architecture
JMS	Java Message Service
LAN	Local Area Network
MB	Megabyte
Mhz	Megahertz
MQAA	MQSeries Application Adapter
MQI	Message Queue Interface
MQSI	MQSeries Integrator
MQWF	MQSeries Workflow
ORB	Object Request Broker
RAM	Random-access Memory

Acronym	Description
SFA	Student Financial Assistance
SNA	Systems Network Architecture
SQL	Structured Query Language
TCP	Transmission Control Protocol
VDC	Virtual Data Center
WAN	Wide Area Network
XML	Extensible Markup Language